

Kapitel 3: Sortieren

Elementare Sortieralgorithmen

Heaps und Heapsort

Quicksort

Eine untere Schranke für das Sortierproblem

Counting-, Radix- und Bucketsort

Algorithmen für Auswahlprobleme

**Folien nur für den eigenen Gebrauch, Weiterleitung
an Dritte und Online-Stellen nicht erlaubt.**

3.1 Elementare Sortieralgorithmen

- Problem:
 - Geg. Menge von Datensätzen s_i mit Schlüsseln k_i .
 - Geg. totale Ordnungsrelation \leq
 - Ges.: Permutation π mit: $k_{\pi(1)} \leq k_{\pi(2)} \leq \dots \leq k_{\pi(n)}$
- partielle Ordnung R :
 - reflexiv: es gilt $a R a$
 - antisymmetrisch: $a R b$ und $b R a$ folgt $a = b$
 - transitiv: $a R b$ und $b R c$ folgt $a R c$
- totale Ordnung R :
 - R ist partielle Ordnung
 - für alle a, b gilt: $a R b$ oder $b R a$
- Bsp: Ordnung auf der Menge der Menschen
 - $R = \text{,ist Nachfahre von'}$ partiell (und nicht reflexiv)
 - $R = \text{,hat größere Ausweisnummer'}$ nicht reflexiv!
 - $R = \text{,hat keine kleinere Ausweisnummer'}$ total

Elementare Sortieralgorithmen

■ Für alle Sortieralgorithmen

■ Datentyp:

struct item

```
{      key : integer  
      info: Grundtyp  
}
```

■ Eingabe: A: array[1...N] of item

■ Ausgabe: A mit vertauschten Einträgen, so dass

$A[i].key \leq A[i+1].key$ für $1 \leq i < N$ gilt.

■ Laufzeitanalyse:

- ◆ Zugriffsoperation c_Z
- ◆ Vergleichsoperationen c_V
- ◆ Datenaustauschoperationen c_D

Sortieren durch Auswahl

Idee:

Iteriere von 1 bis N-1:

im i-ten Durchlauf: Schreibe das kleinste Element aus A[i...n] an Position A[i]

```
SELECTION-SORT (A)
```

```
for i = 1 to A.length-1
```

```
// suche kleinstes Element aus A[i..N]
```

```
min = i
```

```
for j = i+1 to A.length
```

```
if( A[j] < A[min] ) min= j
```

```
// vertausche A[i] und A[min]
```

```
SWAP (A[min], A[i])
```

```
SWAP (a, b)
```

```
t = a; a= b; b = t
```

- Laufzeit: $T(N) = \Theta(N^2)$

- nur $O(N)$ Datenaustausche

index:	1	2	3	4	5	6	7	8
A:	3	8	14	2	9	23	1	4
	↑ i						↑ min	
A:	1	8	14	2	9	23	3	4
	↑ i			↑ min				
A:	1	2	14	8	9	23	3	4
							:	
							:	

Sortieren durch Einfügen

Idee:

Iteriere von 2 bis N:

im j-ten Durchlauf: Füge das j-te Element in die sortierte Folge A[1... j-1] ein

```
INSERTION-SORT (A)
for j = 2 to A.length
    key = A[j]
// setzte A[j] sortiert in A[1..j-1] ein
    i = j-1
    while i>0 and A[i] > key
        A[i+1] = A[i]
        i = i-1
    A[i+1] = key
```

index:	1	2	3	4	5	6	7	8
A:	3	8	14	2	9	23	1	4
i	↑			↑				
j				↑				
A:	2	3	8	14	9	23	1	4
i		↑			↑			
j				↑				
A:	2	3	8	9	14	23	1	4
							↑	
							↑	

Laufzeit (im Worst Case): $T(N) = \Theta(N^2)$

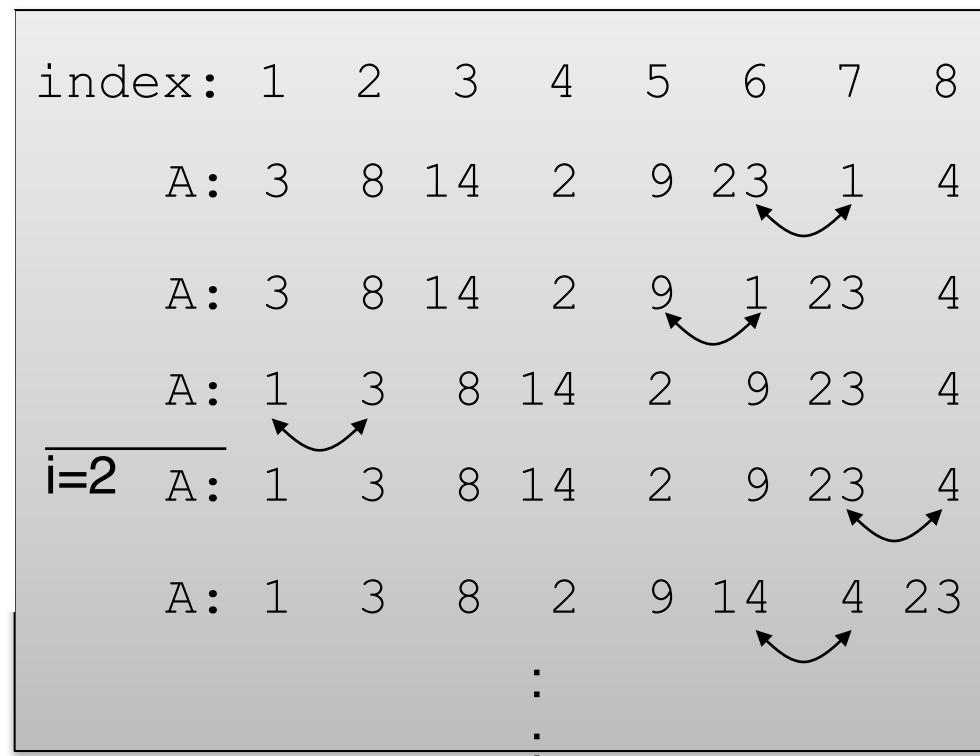
Bubblesort

Idee:

■ Iteriere solange, bis keine Vertauschung mehr durchgeführt wird:

im i-ten Durchlauf (for-Schleife): bringe Minimum von A[i...N] an Position A[i] durch sukzessives Vertauschen benachbarter Elemente

```
BUBBLESORT(A)
for i = 1 to A.length
    noswap = TRUE
    for j = A.length downto i+1
        if A[j] < A[j-1]
            SWAP( A[j], A[j-1] )
            noswap = FALSE
    if( noswap ) break
```



■ Laufzeit (im Worst Case): $T(N) = \Theta(N^2)$

Mergesort

Idee: Divide & Conquer

- Teile die Folge in zwei Teilfolgen $L = A[1 \dots \lfloor N/2 \rfloor]$, $R = A[\lfloor N/2 \rfloor + 1 \dots N]$
- Sortiere L und R
- Verschmelze sortierte Folgen L und R zu einer sortierten Folge

```
MERGE-SORT(A, l, r)
```

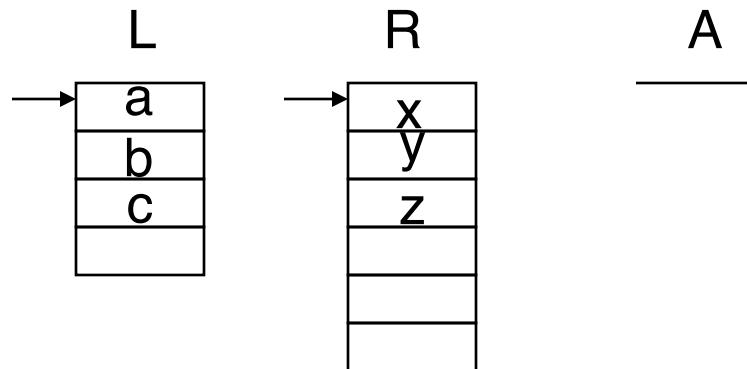
```
if l < r
    m = floor((l+r)/2)
    MERGE-SORT(A, l, m)
    MERGE-SORT(A, m+1, r)
    MERGE(A, l, m, r)
```

								L	R							
								index:	1	2	3	4	5	6	7	8
Divide:																
A: 3 8 14 2 9 23 1 4																
Conquer:																
A: 2 3 8 14 1 4 9 23																
Merge:																
A: 1 2 3 4 8 9 14 23																

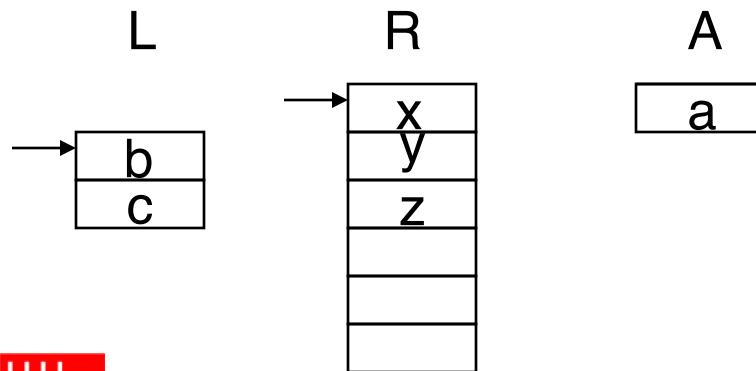
2-Wege Merge-Operation

Merge-Operation (mit Listen)

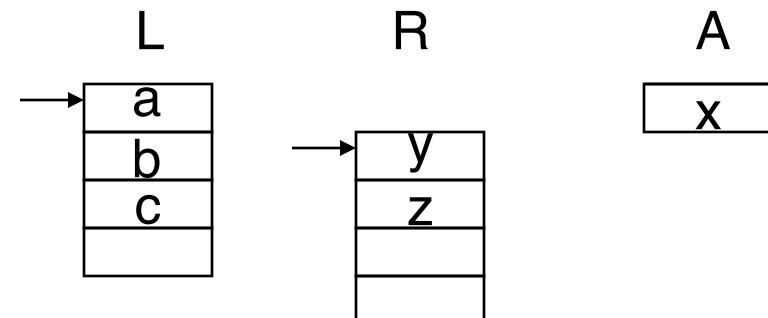
- geg.: zwei aufsteigend sortierte Listen L und R
- ges.: eine aufsteigend sortierte Liste A aus L und R



Fall 1: $L.\text{head} < R.\text{head}$
oder $\text{EMPTY}(R)$



Fall 2: $L.\text{head} > R.\text{head}$
oder $\text{EMPTY}(L)$



2-Wege Merge-Operation

■ Listenoperationen:

- INIT: initialisiere leere Liste
- head: erstes Element der Liste
- APPEND: einfügen am Ende der Liste
- EMPTY: wahr, falls Liste leer
- TAIL: Liste ohne erstes Element

■ MERGE-LISTS (L, R : list)

```
A = INIT()  
while not (EMPTY(L) and EMPTY(R))  
    if EMPTY(R) or L.head ≤ R.head  
        APPEND(A, L.head)  
        L = TAIL(L)  
    elseif EMPTY(L) or L.head ≥ R.head  
        APPEND(A, R.head)  
        R = TAIL(R)  
return A
```

Laufzeit: $O(|L|+|R|)$

2-Wege Merge-Operation mit Arrays

```

procedure merge (var a : sequence; l, m, r : integer);
{verschmilzt die beiden sortierten Teilfolgen a[l]...a[m]
 und a[m + 1]...a[r] und speichert sie in a[l]...a[r]}
var
  b : sequence; {Hilfsfeld zum Verschmelzen}
  h, i, j, k : integer;
begin
  i := l; {inspiziere noch a[i] bis a[m] der ersten Teilfolge}
  j := m + 1; {inspiziere noch a[j] bis a[r] der zweiten Teilfolge}
  k := l; {das nächste Element der Resultatfolge ist b[k]}
  while (i ≤ m) and (j ≤ r) do
    begin {beide Teilfolgen sind noch nicht erschöpft}
      if a[i].key ≤ a[j].key
        then {übernimm a[i] nach b[k]}
          begin
            b[k] := a[i];
            i := i + 1
          end
        else {übernimm a[j] nach b[k]}
          begin
            b[k] := a[j];
            j := j + 1
          end;
      k := k + 1
    end;
  if i > m
  then {erste Teilfolge ist erschöpft; übernimm zweite}
    for h := j to r do b[k + h - j] := a[h]
  else {zweite Teilfolge ist erschöpft; übernimm erste}
    for h := i to m do b[k + h - i] := a[h];
  {speichere sortierte Folge von b zurück nach a}
  for h := l to r do a[h] := b[h]
end

```

■ Liste L: $a[l \dots m]$ Liste R: $a[m+1 \dots r]$

■ Liste A: zunächst in b,
wird abschließend kopiert nach a

■ i: index von L.head

■ j: index von R.head

■ k: index von A.rear

■ ... APPEND(...)

■ ... TAIL(...)

■ andere Schleifenstruktur:

◆ Teil 1: beide Listen nicht leer

◆ Teil 2: Liste L leer

◆ Teil 3: Liste R leer

■ Kopieren von b nach a

Laufzeit Mergesort

■ Mergesort

1. Teile die Folge in zwei Teilstücke

$$L = A[1 \dots \lceil n/2 \rceil], R = A[\lceil n/2 \rceil + 1 \dots n]$$

$O(1)$

2. Sortiere L und R

$2 * T(N/2)$

3. Verschmelze sortierte Folgen L und R zu einer sortierten Folge

$O(N)$

■ Laufzeit: $T(N) = 2 * T(N/2) + cN = O(N \log N)$

■ Speicherbedarf:

- Array A: $c * N$
- Zwischenspeicherung in zweitem Array: $c * N$
- doppelter Speicherbedarf!

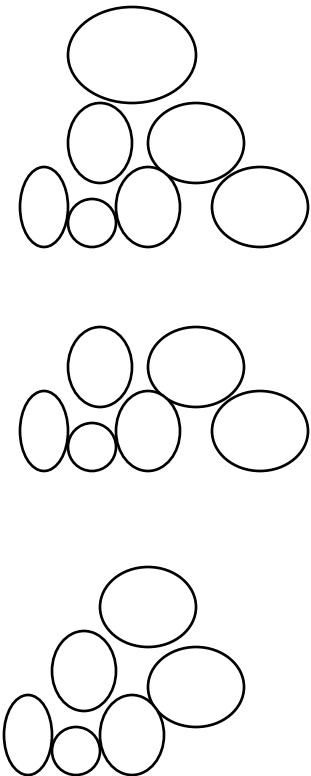
■ k-Wege Mergesort: Teile und verschmelze k Teilstücke

Laufzeit: $T(N) = k * T(N/k) + c k N = O(N \log N)$

($k * N$, da in jeder Iteration das Minimum aus k Listenköpfen bestimmt werden muss)

3.2 Heaps und Heapsort

- Sortieren durch Auswahl (siehe 3.1)
 - das kleinste Element wird ausgewählt und an die bereits sortierte Liste angehängt
 - Wie gestaltet man den Auswahlsschritt?
- Neue Datenstruktur: Heap
 - „Objekte liegen auf einer Halde, ein Objekt ist immer mindestens so groß wie die darunter liegenden Objekte“ (Heap-Bedingung)
 - Operationen:
 - ◆ BUILD-MAX-HEAP: baue eine Halde aus einer Menge von Elementen
 - ◆ HEAP-EXTRACT-MAX: nehme das größte Element von der Halde
 - ◆ MAX-HEAPIFY: stelle die Heap-Bedingung nach Entnahme wieder her



Der Heapsort-Algorithmus

■ HEAPSORT-WITH-HEAP (A)

```
// Heapsort mit HEAP-Datenstruktur  
H = INIT()  
for i = 1 to A.length  
    HEAP-INSERT(H, A[i] )  
for i = A.length downto 1  
    A[i] = HEAP-EXTRACT-MAX(H)
```

■ HEAPSORT (A)

```
// Heap mit i Elementen steht im  
// Array A[1..i], A[1] enthält das  
// Maximum  
BUILD-MAX-HEAP (A)  
A.heap_size = A.length  
for i = A.length downto 2  
    SWAP( A[i], A[1] )  
    A.heap_size = A.heap_size - 1  
MAX-HEAPIFY (A, 1)
```

■ Laufzeit:

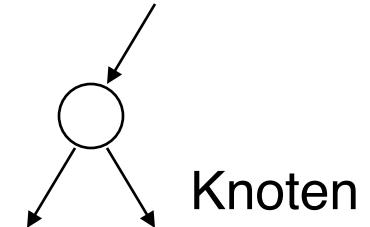
$$\blacksquare T(N) = O(T_{\text{build}}(N) + N T_{\text{xmax}}(N))$$

- füge alle Elemente in den Heap ein (Zeit $T_{\text{build}}(N)$)
- entnehme das größte Element (und stelle die Heap-Eigenschaft wieder her) (Zeit $T_{\text{xmax}}(N)$)

Der Heap

■ Der Heap basiert auf einer *binären Baumstruktur*:

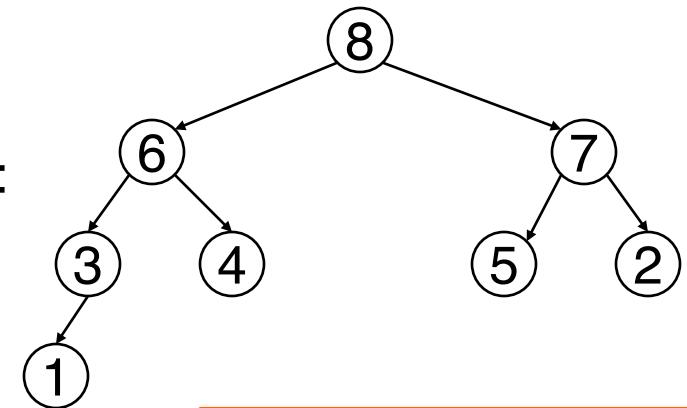
- ◆ jeder Knoten i hat bis zu zwei Nachfolger $\text{LEFT}(i)$, $\text{RIGHT}(i)$
- ◆ alle bis auf ein Knoten haben genau einen Vorgänger $\text{PARENT}(i)$
- ◆ jeder Knoten speichert ein Element $A[i]$



■ Heap-Eigenschaft:

Max-Heap: \forall Knoten i außer der Wurzel:

$$A[\text{PARENT}(i)] \geq A[i]$$



Min-Heap: \forall Knoten i außer der Wurzel:

$$A[\text{PARENT}(i)] \leq A[i]$$

Links vollständigkeit
diskutieren

Der Heap

- Ein Heap kann durch ein Array repräsentiert werden:

- PARENT(i) = i div 2
- LEFT(i) = 2 * i
- RIGHT(i) = 2 * i + 1

- Begriffe: Sei A ein Heap

- Höhe von A: max Anzahl Knoten

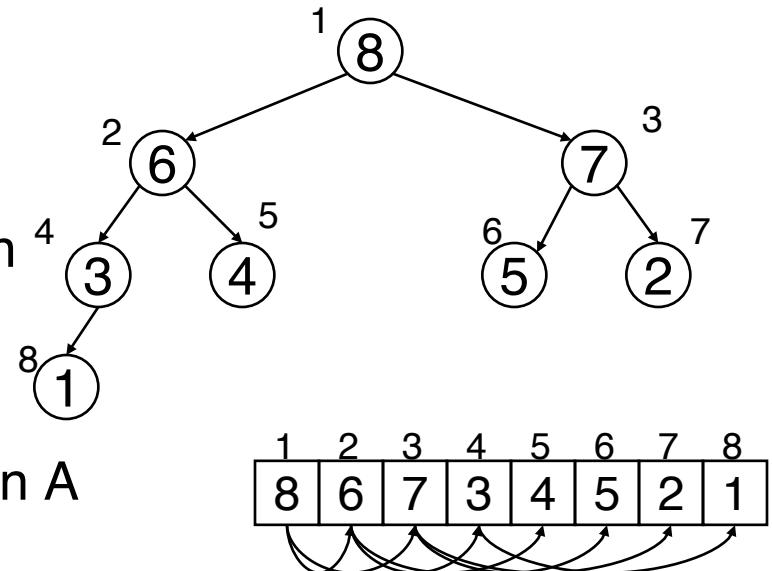
auf dem direktem Weg zu
einem Blatt

- A.heap_size: Anzahl Elemente in A

- Weitere Eigenschaften:

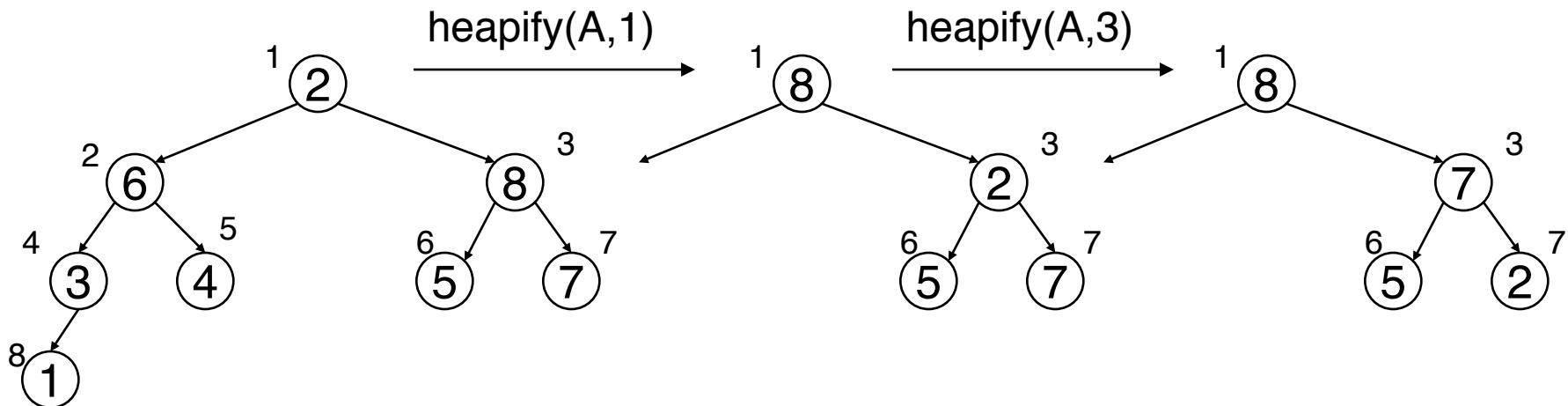
- Ein Heap der Größe N hat max. Höhe: $\lfloor \log_2 N \rfloor$

- auf Ebene i liegen 2^i Knoten



Die MAX-HEAPIFY-Operation

- Heap-Eigenschaft (top-down): Sei A ein Heap, dann gilt
 - $A[i] \geq A[LEFT(i)]$ und $A[i] \geq A[RIGHT(i)]$
- Heapify (Versickern):
 - Annahme: A erfüllt Heap-Eigenschaft bis auf in einem Teilbaum unter Position i
 - MAX-HEAPIFY(A,i): repariert den Heap (stellt die Heap-Eigenschaft wieder her)



HEAPIFY:
- bestimme direkten Nachfolger c mit max. Wert $A[c]$
- vertausche $A[i]$ mit $A[c]$
- HEAPIFY(A, c)

Die MAX-HEAPIFY-Operation

■ MAX-HEAPIFY (A, i)

```
if( LEFT(i) ≤ A.heap_size and  
    A[LEFT(i)] > A[i] )  
    c = LEFT(i)  
else c = i  
  
if( RIGHT(i) ≤ A.heap_size and  
    A[RIGHT(i)] > A[c] )  
    c = RIGHT(i)  
  
if c ≠ i  
    SWAP(A[i], A[c])  
    MAX-HEAPIFY( A, c)
```

c: Index des größten Elements aus der Menge $i, \text{left}(i), \text{right}(i)$ innerhalb von A

heap_size[A]: Größe des Heaps, nicht die Anzahl der Elemente des Arrays!

$c = i$: Heap-Eigenschaft war nicht verletzt

$c \neq i$: vertausche Elemente c und i

■ Laufzeit:

$$T_{\text{heapify}}(n) = T_{\text{heapify}}(2n/3) + \Theta(1)$$

Master-Theorem, Fall 2:

$$T_{\text{heapify}}(n) = O(\log(n))$$



Laufzeit der MAX-HEAPIFY-Operation

■ Rekurrenzgleichung:

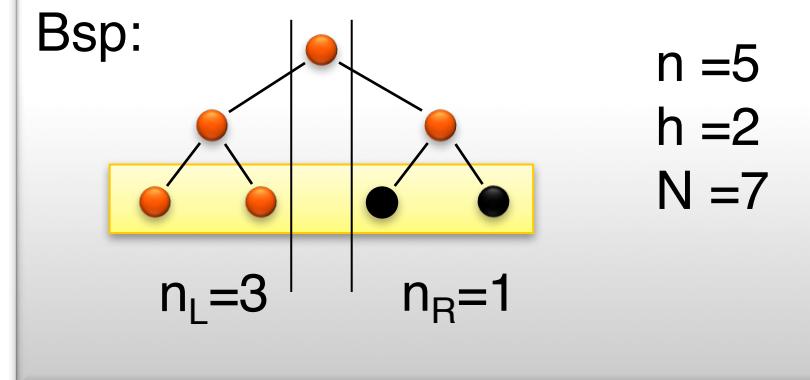
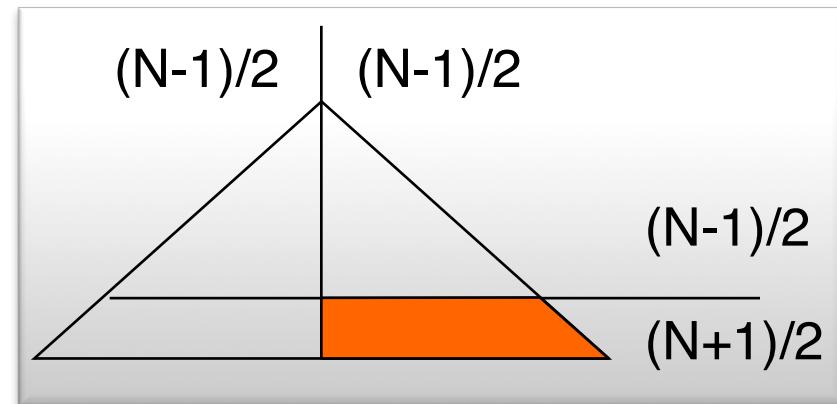
Im Worst-Case gilt: $T_{\text{heapify}}(n) = T_{\text{heapify}}(2n/3) + \Theta(1)$

■ Beweis:

■ Betrachte zunächst einen vollständigen binären Baum B der Höhe $h = \lfloor \log_2 N \rfloor$:

Sei N die Anzahl der Knoten von B, dann gilt:

- ◆ B hat $N = 2^{h+1} - 1$ Knoten
- ◆ B hat $(N+1)/2$ Blätter
- ◆ B hat $(N-1)/2$ Knoten in jedem Teilbaum unter der Wurzel



Laufzeit der MAX-HEAPIFY-Operation

■ Beweis (Fortsetzung):

Worst Case: MAX-HEAPIFY verzweigt in den größeren Teilbaum (links) mit insgesamt n_L Knoten: $T_{\text{heapify}}(n) = T_{\text{heapify}}(n_L) + \Theta(1)$

Wie groß ist n_L in Bezug zur Gesamtzahl der Knoten n im Worst Case?

$$n_L = \frac{N-1}{2}$$

$$n_R = \frac{N-1}{2} - \frac{N+1}{4} = \frac{N-3}{4}$$

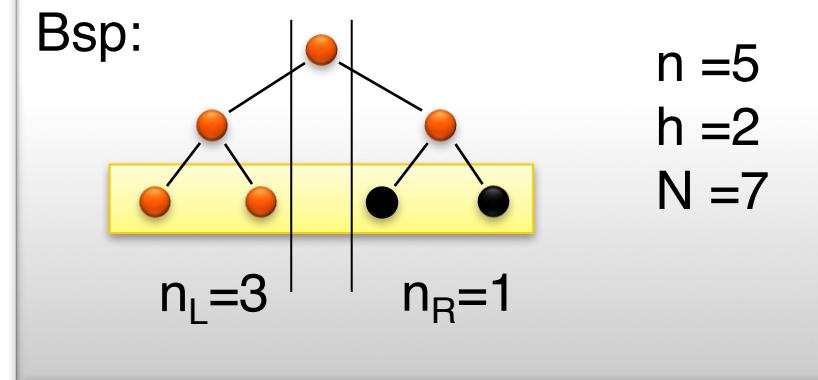
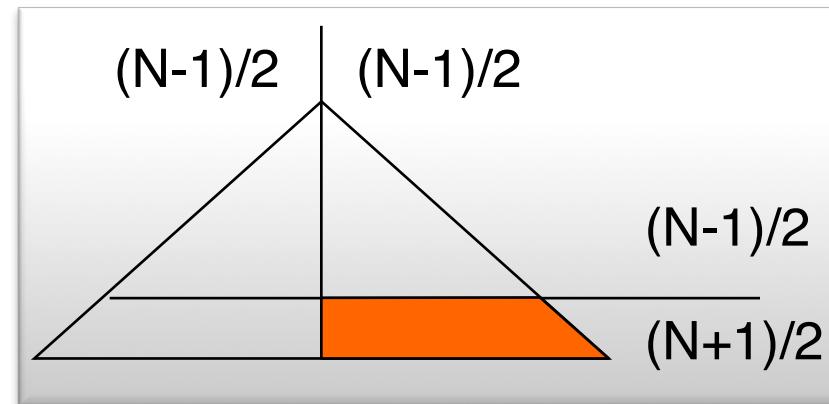
$$n = 1 + n_L + n_R$$

$$= 1 + \frac{N-1}{2} + \frac{N-1}{2} - \frac{N+1}{4}$$

$$= \frac{3N-1}{4}$$

$$\Leftrightarrow N = \frac{4n+1}{3}$$

$$\Rightarrow n_L = \frac{N-1}{2} = \frac{\frac{4n+1}{3}-1}{2} = \frac{2}{3}n - \frac{1}{3} \leq \frac{2}{3}n$$



Die HEAP-EXTRACT-MAX-Operation

■ HEAP-EXTRACT-MAX:

Entnehme das maximale Element aus dem Heap

■ HEAP-EXTRACT-MAX (A)

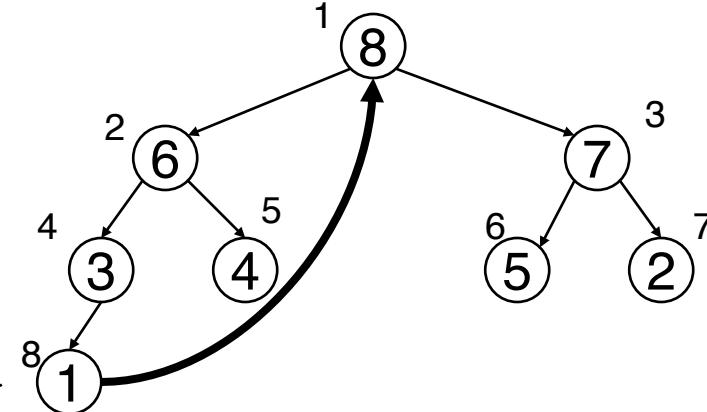
```
m = A[1]
```

```
A[1] = A[A.heap_size]
```

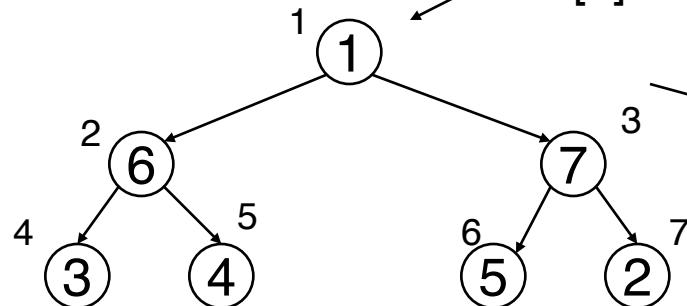
```
A.heap_size = A.heap_size - 1
```

```
MAX-HEAPIFY (A, 1)
```

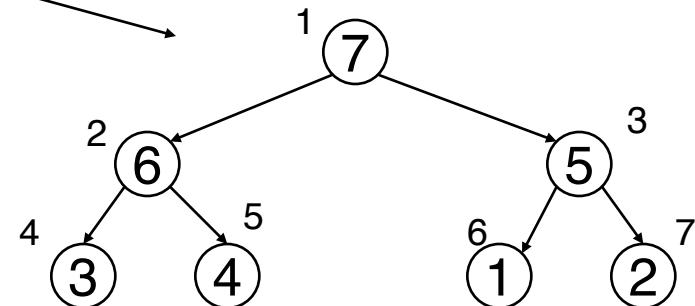
```
return m
```



$A[1] \leftarrow A[A.heap_size]$



heapify(A, 1)



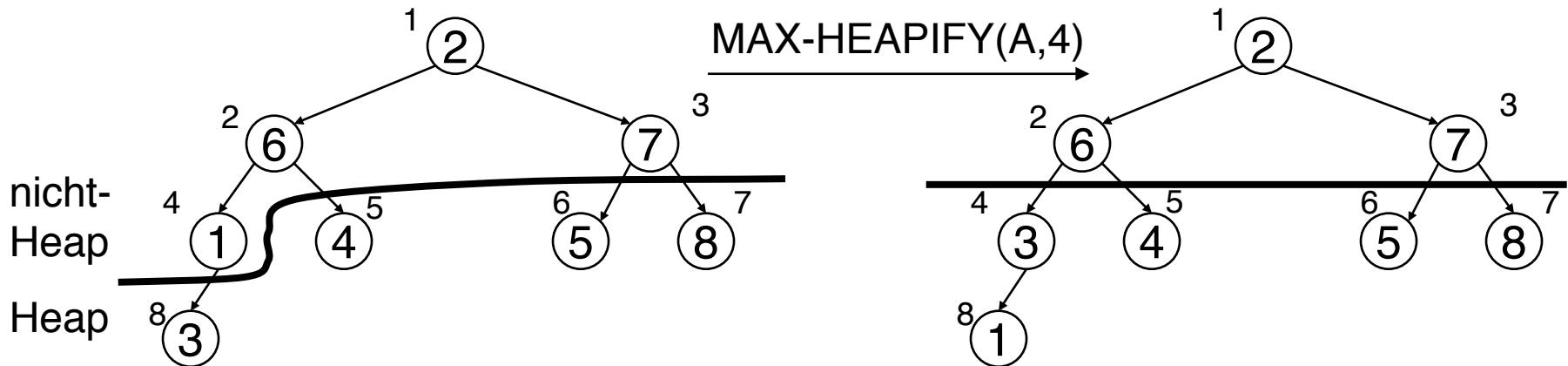
Die BUILD-MAX-HEAP-Operation

BUILD-MAX-HEAP:

- Eingabe: unsortiertes Array A
- Ausgabe: Heap (als Array in A)

Idee:

- Elemente $\lfloor A.\text{heap_size}/2 \rfloor + 1$ bis $A.\text{heap_size}$ (die Blätter) sind ein-elementige Heaps
- Verwende MAX-HEAPIFY um den Heap von unten nach oben aufzubauen



Die BUILD-MAX-HEAP-Operation

■ BUILD-MAX-HEAP (A)

```
A.heap_size = A.length  
for i = ⌊A.heap_size/2⌋ downto 1  
    MAX-HEAPIFY (A, i)
```

n/2 Durchläufe
 $O(\log n)$

■ Korrektheit von BUILD-MAX-HEAP:

Schleifeninvariante: $\forall k \in \{i+1, \dots, n=A.length\}$: A[k] ist die Wurzel eines Max-Heaps.

■ Initialisierung:

- ◆ Knoten $A[\lfloor n/2 \rfloor + 1], A[\lfloor n/2 \rfloor + 2], \dots, n$ sind Blätter und erfüllen somit die Max-Heap-Eigenschaft.

■ Fortsetzung: Für Knoten i gilt:

- ◆ LEFT(i) und RIGHT(i) sind Max-Heaps wg. der Schleifeninvariante
- ◆ MAX-HEAPIFY(A,i) stellt somit die Max-Heap-Eigenschaft her.

■ Terminierung:

- ◆ Schleife terminiert mit $i=0$, damit in $A[1]$ die Wurzel eines Max-Heap.

Die BUILD-MAX-HEAP-Operation

■ Laufzeit von BUILD-MAX-HEAP(A)

Sei $n = A.length$, offensichtlich gilt $T_{\text{build}}(n) = O(n \log n)$.

■ Gilt auch $T_{\text{build}}(n) = \Theta(n \log n)$?

■ Laufzeit ist proportional zur akkumulierten Laufzeit der MAX-HEAPIFY-Operationen

■ MAX-HEAPIFY für einen Knoten mit Höhe h : $O(h)$

■ Höhe variiert von $h=0$ bis $\lfloor \log n \rfloor$

■ Anzahl Knoten mit Höhe h : $\lceil n / 2^{h+1} \rceil$

(Anzahl Knoten halbiert sich von Ebene zu Ebene)

$$T(n) = c \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil h = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) = O(n)$$

$$\text{mit } \sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h = \frac{1/2}{\left(1 - 1/2\right)^2} = 2$$

(Ableitung der geom.
Reihe, siehe Cormen A.8)

Prioritätswarteschlangen (Priority Queues)

■ Priority Queue:

■ Ziel: Warteschlange mit Prioritäten

■ Operationen:

- ◆ HEAP-INSERT(A, x): füge ein Element x in die Warteschlange A ein
- ◆ HEAP-EXTRACT-MAX(A): entnehme das Element mit max. Priorität
- ◆ HEAP-INCREASE-KEY(A, i, k): erhöhe den Wert von Schlangenelement Nummer i auf k , $k > A[i]$
- ◆ HEAP-DECREASE-KEY(A, i, k): verringert den Wert von Schlangenelement Nummer i auf k , $k < A[i]$

■ Realisierung: durch einen Heap

- ◆ HEAP-EXTRACT-MAX-Operation: wie zuvor, Laufzeit $O(\log N)$
- ◆ HEAP-DECREASE-KEY-Operation: wie MAX-HEAPIFY, Laufzeit $O(\log N)$

Achtung:

- Typo im Cormen S. 135: INCREASE-KEY(S, i, k), nicht (S, x, k)!
- Im Gegensatz zum Buch ist hier DECREASE-KEY() in einem Max-Heap gemeint.

Die HEAP-INCREASE-KEY-Operation

■ HEAP-INCREASE-KEY (A, i, k)

```
if(  $k < A[i]$  ) error „Increase-Error“
```

```
else
```

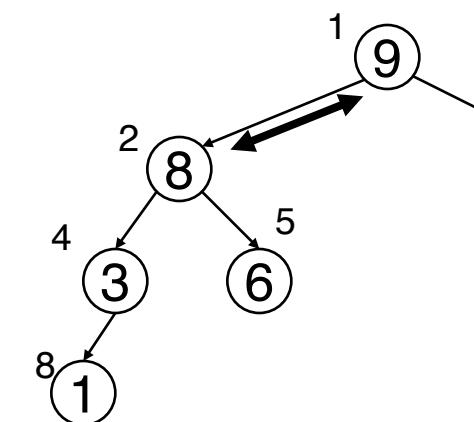
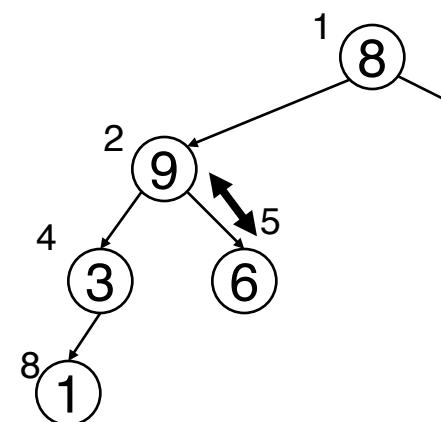
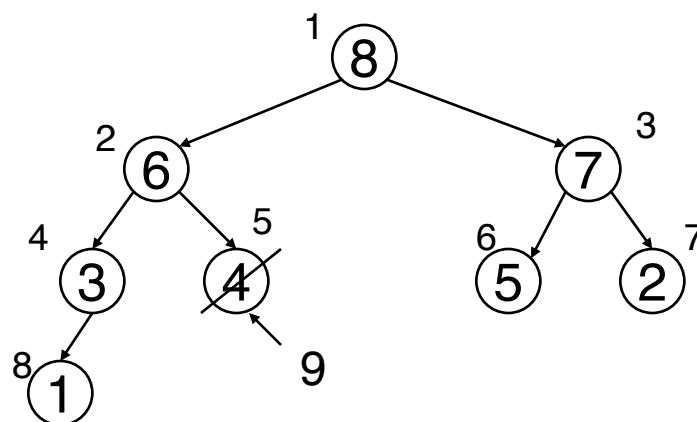
```
   $A[i] = k$ 
```

```
  while  $i > 1$  and  $A[\text{parent}(i)] < A[i]$  //  $\text{parent}(i)$  entspr. hier  $(i \text{ div } 2)$ 
```

```
    SWAP( $A[i], A[\text{parent}(i)]$ )
```

```
     $i = \text{parent}(i)$ 
```

■ HEAP-INCREASE-KEY funktioniert wie heapify in umgekehrter Richtung:



Die INSERT-Operation

■ HEAP-INSERT (A, x)

```
A.heap_size = A.heap_size + 1
```

```
A[A.heap_size] = -∞
```

```
HEAP-INCREASE-KEY (A, A.heap_size, x)
```

■ Mit einem Heap kann eine Priority-Queue mit Laufzeit $O(\log N)$ für die Operationen insert, delete, increase, decrease, extract-max realisiert werden.

■ Anmerkung:

Zum Auffinden eines Wertes x benötigt man eine zusätzliche Datenstruktur, z.B.

■ **HEAP-INCREASE-KEY2(A, x, k):**

Problem: Finde i mit $A[i] = x$

3.3 Quicksort

- Idee: Divide & Conquer (T. Hoare, 1960)
 - Teile die Folge in zwei Teilfolgen $A[1\dots q-1]$ und $A[q+1\dots n]$ mit:
 $A[i] \leq A[q] \quad \forall i < q$ und $A[i] \geq A[q] \quad \forall i > q$ (*)
 - Sortiere die Teilfolgen $A[1\dots q-1]$ und $A[q+1\dots n]$
 - (Kombination der Teilfolgen nicht nötig aufgrund von (*))

```
QUICKSORT( A, l, r )
if l < r
    q = PARTITION(A, l, r) // Bestimmt q und stellt Bedingung (*) her
    QUICKSORT(A, l, q-1)
    QUICKSORT(A, q+1, r)
```

- initialer Aufruf: `QUICKSORT(A, 1, A.length)`

Partition-Funktion

■ Partition-Funktion (Hoare-Zerlegung)

- geg: ein Array A[l...r]
- ges: q mit $l \leq q \leq r$, umsortiertes Array A mit
 $A[i] \leq A[q] \quad \forall i < q$ und $A[i] \geq A[q] \quad \forall i > q$ // A[q]: *Pivot-Element*

```
PARTITION_HOARE(A, l, r )
// verwende A[r] als Pivot-Element
1 i = l-1; j = r;
2 while i < j
3   repeat i = i+1 until A[i] ≥ A[r]
4   repeat j = j-1 until A[j] ≤ A[r]
5   if( i < j ) SWAP( A[i], A[j] )
6   SWAP( A[i], A[r] )    // schiebt das Pivot-Element in die Mitte
7   return i
```

Partition-Funktion

■ Beispiel: Partition-Funktion nach Hoare

	I								r
index:	1	2	3	4	5	6	7	8	
A :	3	8	14	2	9	23	1	4	Pivot-Element
nach 4.:	A :	3	8	14	2	9	23	1	4
nach 5.:	A :	3	1	14	2	9	23	8	4
nach 4.:	A :	3	1	2	14	9	23	8	4
nach 5.:	A :	3	1	2	14	9	23	8	4
nach 4.:	A :	3	1	2	14	9	23	8	4
nach 7.:	A :	3	1	2	4	9	23	8	14
		\leq				\geq			

i und j sind übereinander
hinweggelaufen

Partition-Funktion

- Alternative Prozedur mit nur einer Schleife:

- PARTITION (A, l, r)

```
// verwende A[r] als Pivot-Element
```

```
i = l-1
```

```
for j = l to r-1
```

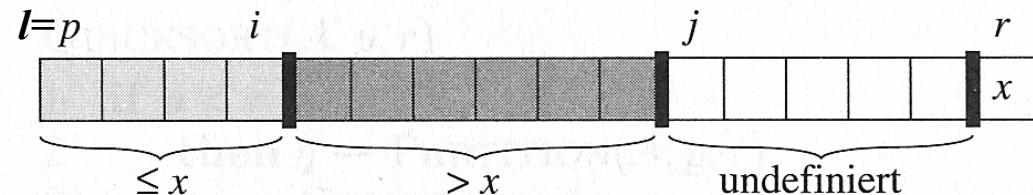
```
  if A[j] ≤ A[r]
```

```
    i = i+1
```

```
    SWAP( A[i], A[j] )
```

```
SWAP( A[i+1], A[r] )
```

```
return i+1
```



- Funktionsweise:

PARTITION zerlegt A in vier (ggf. leere) Bereiche:

1. für $l \leq k \leq i$ gilt: $A[k] \leq A[r]$
2. für $i < k < j$ gilt: $A[k] > A[r]$
3. für $j \leq k < r$ gilt: $A[k] ? A[r]$
4. für $k = r$ gilt: $A[k] = A[r]$

Partition-Funktion

■ PARTITION (A, l, r)

// verwende A[r] als Pivot

i = l-1

for j = l **to** r-1

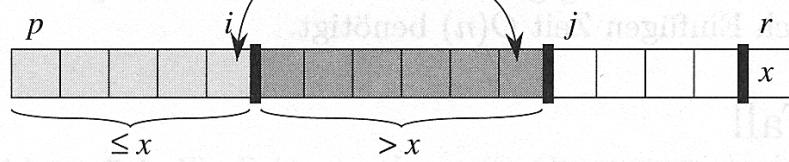
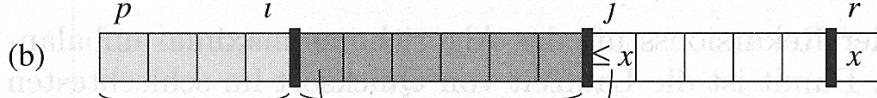
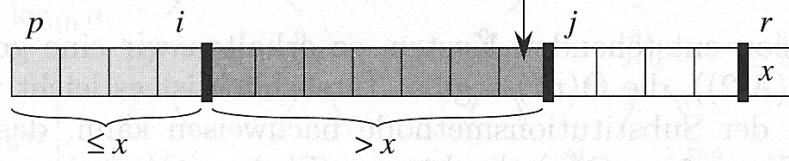
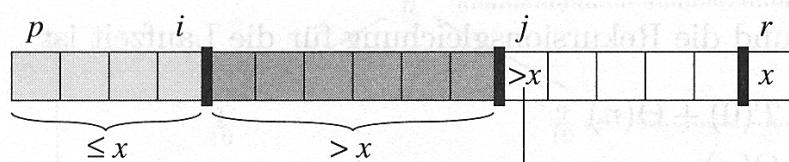
if A[j] \leq A[r]

i = i+1

SWAP(A[i], A[j])

SWAP(A[i+1], A[r])

return i+1



■ Korrektheit: (informell)

■ Fall 1: A[j] $>$ A[r] : Verlängere Bereich 2 um ein Feld

■ Fall 2: A[j] \leq A[r] : Vertausche A[i] und A[j],

vergrößere Bereich 1 um ein Feld,

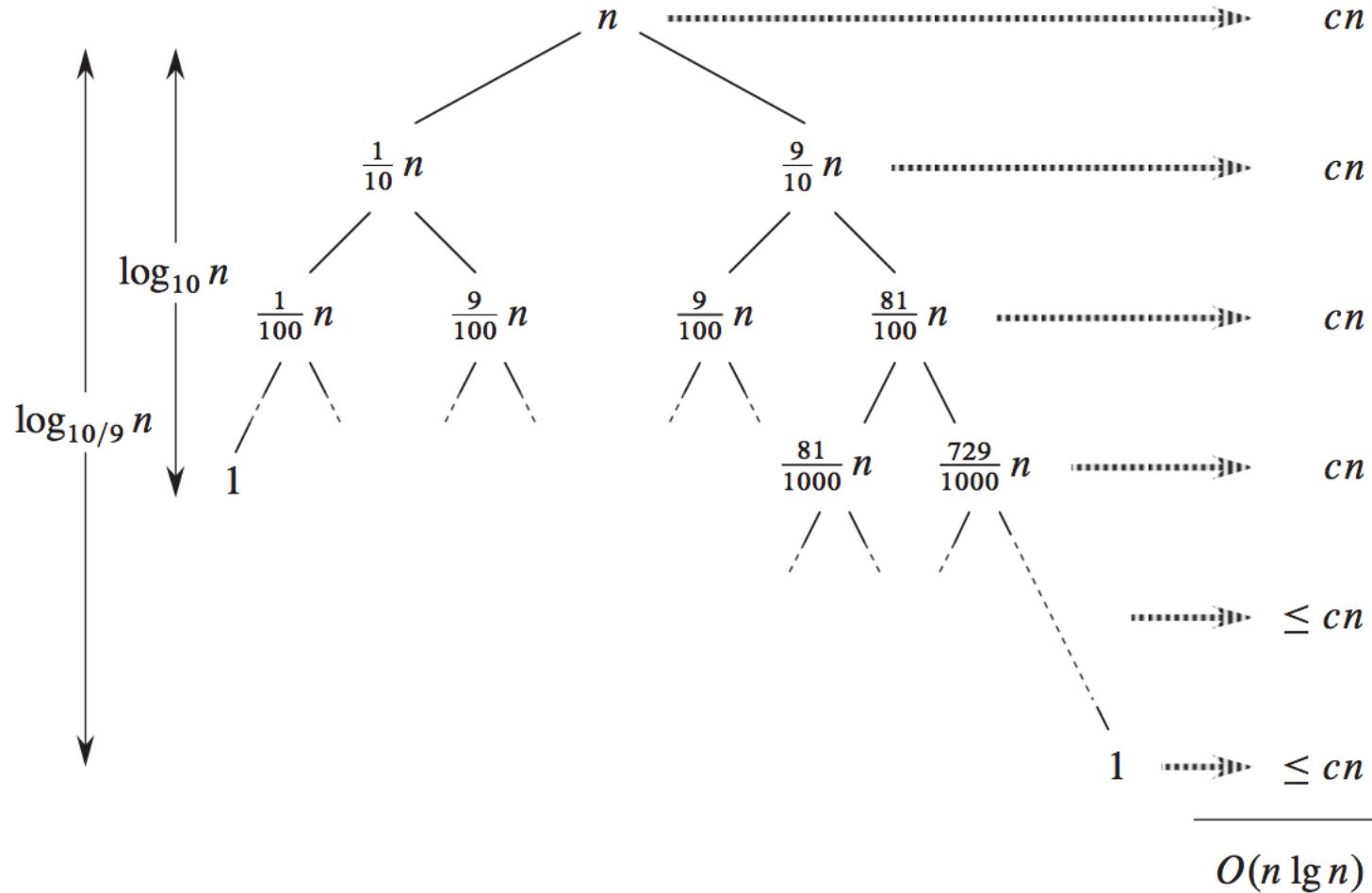
verlängere Bereich 2 um ein Feld

Analyse von Quicksort

- Rekurrenzgleichung für Quicksort:
 - Sei q die Position des Pivot-Elements, dann gilt
$$T(N) = T(q - 1) + T(N - q) + c N$$
- Annahme: Pivot-Element ist immer das Maximum der sortierten Teilfolge, d.h. $q = N$:
$$T(N) = T(N-1) + T(0) + c * N = c*(N + N-1 + N-2 + \dots + 1) = \Theta(N^2)$$
- Annahme: Pivot-Element liegt in der Mitte der sortierten Teilfolge, d.h. $q = \lceil N/2 \rceil$
$$T(N) = T(\lceil N/2 \rceil - 1) + T(N - \lceil N/2 \rceil) + c N \cong 2 T(N/2) + c N = \Theta(N \log N)$$
- Annahme: Pivot-Element liegt bei $q = aN$ für ein konstantes $0,5 < a < 1$
 - $T(N) = T(\lceil a N \rceil) + T(\lceil (1-a)N \rceil) + c N$
 - Rekursionsbaum mit maximal $-\log_a N = \log_{1/a} N$ Ebenen
 - Laufzeit pro Ebene des Rekursionsbaums: cN
 - $T(N) = O(N \log N)$

Analyse von Quicksort

■ Beispiel für $a = 0,9$:



Der randomisierte Quicksort

■ Randomisierter Algorithmus:

Algorithmus, der unter Verwendung von Zufallszahlen operiert.

■ ACHTUNG: Randomisierte Algorithmen sind (i.d.R.) deterministisch (d.h. sie liefern immer das gleiche Ergebnis)!

■ RAND-PARTITION (A, l, r)

```
i = RANDOM(l, r)      // wählt eine ganzzahlige Zufallszahl aus [l,r]
SWAP( A[i], A[r] )
return PARTITION(A, l, r)
```

■ RAND-QUICKSORT (A, l, r)

```
if l < r
    q = RAND-PARTITION(A, l, r)
    RAND-QUICKSORT(A, l, q-1)
    RAND-QUICKSORT(A, q+1, r)
```

■ Erwartete Laufzeit von RAND-QUICKSORT ist unabhängig von der Vorsortierung

Analyse des randomisierten Quicksort

- **Theorem:** Angenommen, alle Werte sind voneinander verschieden, dann ist die *erwartete Laufzeit* von Quicksort $O(N \log N)$.

- **Beweisalternative 1:** (Mittelung über die Rekurrenzgleichung)

$$T_{\text{exp}}(N) = \left[\frac{1}{N} \sum_{q=1}^N T(q-1) + T(N-q) \right] + cN \leq \left[\frac{2}{N} \sum_{q=1}^{\lceil N/2 \rceil} T(q) \right] + cN \stackrel{?}{=} O(N \log N)$$

- **Beweisalternative 2:** (Erwartete Anzahl Vergleiche)

- Laufzeit von Quicksort wird durch PARTITION dominiert.
- Laufzeit von PARTITION ist proportional zur Anzahl Vergleiche.
- Sei X die Anzahl Vergleiche, dann ist $T(N, X) = O(N + X)$.
- Wie hoch ist die erwartete Anzahl Vergleiche $E[X]$?
 - Sei X_{ij} das Ereignis, dass $A[i]$ mit $A[j]$ verglichen wird.
 - Wenn $X_{ij}=1$, ist entweder $A[i]$ oder $A[j]$ Pivotelement, jeder Paarvergleich findet nur ein mal statt:

$$E[X] = E \left[\sum_{i=1}^{N-1} \sum_{j=i+1}^N X_{ij} \right] = \sum_{i=1}^{N-1} \sum_{j=i+1}^N E[X_{ij}] = \sum_{i=1}^{N-1} \sum_{j=i+1}^N \Pr \{X_{ij} = 1\}$$

Analyse des randomisierten Quicksort

- Wie hoch ist die Wahrscheinlichkeit, dass $A[i]$ mit $A[j]$ verglichen wird, d.h. $\Pr\{X_{ij}=1\}$?
 - Sei x der Pivotwert, dann gilt für alle $A[i] < x < A[j]$: $\Pr\{X_{ij}=1\} = 0$
 - Für jedes Teil-Array $A[i\dots j]$ gilt: $A[i]$ wird mit $A[j]$ verglichen, genau dann wenn entweder $A[i]$ oder $A[j]$ als Pivotelement gewählt werden.
 - $A[i\dots j]$ enthält $j-i+1$ Elemente, die Wahrscheinlichkeit zur Auswahl des Pivotelements ist gleichverteilt (RAND-PARTITION):
 $\Pr\{X_{ij}=1\} = \Pr\{A[i] \text{ oder } A[j] \text{ ist erstes Pivotelement aus } A[i\dots j]\}$

$$\Pr\{X_{ij} = 1\} = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

$$E[X] = \sum_{i=1}^{N-1} \sum_{j=i+1}^N \Pr\{X_{ij} = 1\} = \sum_{i=1}^{N-1} \sum_{j=i+1}^N \frac{2}{j-i+1} \quad \text{mit } k = j - i$$

$$\begin{aligned} &= \sum_{i=1}^{N-1} \sum_{k=1}^{N-i} \frac{2}{k+1} < \sum_{i=1}^{N-1} \sum_{k=1}^N \frac{2}{k} = \sum_{i=1}^{N-1} 2 \sum_{k=1}^N \frac{1}{k} \leq \sum_{i=1}^{N-1} 2(\log N + 1) \\ &= 2(N-1)(\log N + 1) = O(N \log N) \end{aligned}$$

mit A.7
(harmonische
Reihe)

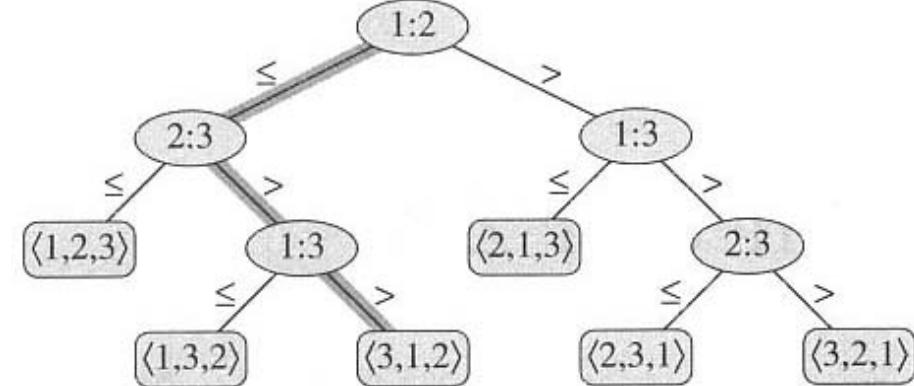
Abschließende Kommentare zu Merge-, Heap- und Quicksort

- **Sortieren durch Einfügen** hat eine Worst-Case Laufzeit von $O(N^2)$, ist allerdings vorteilhaft, wenn die Eingabe nahezu sortiert ist.
- **Mergesort** hat eine Worst-Case Laufzeit von $O(N \log N)$, ist aber kein in-place Sortierverfahren.
- **Mergesort** eignet sich gut für große Datenbestände auf Sekundärspeichern.
- **Heapsort** hat eine Worst-Case Laufzeit von $O(N \log N)$ und sortiert in-place.
- Die Worst-Case Laufzeit von **Quicksort** ist $O(N^2)$, die erwartete Laufzeit des randomisierten Verfahrens $O(N \log N)$. Um eine Worst-Case Laufzeit von $O(N \log N)$ zu erhalten, muß der Median der Eingabefolge in $O(N)$ Zeit bestimmt werden.
- **Quicksort** zeigt in der Praxis ein sehr gutes Laufzeitverhalten (niedrige Konstanten).

Sind $O(N \log N)$ -Sortierverfahren laufzeitoptimal?

3.4 Eine untere Schranke für das Sortierproblem

- Annahme: alle zu sortierenden Elemente sind paarweise verschieden.
- Vergleichende Sortierverfahren:
 - basieren ausschließlich auf Paarvergleiche zwischen den Elementen der Eingabe.
 - Operationen auf Elemente beschränkt auf: $<$, \leq , $=$, \geq , $>$
 - Operationen können auf eine reduziert werden: \leq
- Das Entscheidungsbaummodell:
 - vollständiger binärer Baum:
 - innere Knoten: Vergleich zwischen zwei Elementen $A[i]$ und $A[j]$
 - Blätter: Permutation (Rangfolge) der Elemente



Eine untere Schranke für das Sortierproblem

- **Theorem:** Jedes vergleichende Sortierverfahren benötigt bei n zu sortierenden Datenelementen im Worst-Case $\Omega(n \log n)$ Vergleichsoperationen.
 - Jedes Sortierverfahren führt eine Serie von Vergleichsoperationen aus.
 - diese entspricht einem Pfad von Wurzel zu Blatt in einem entsprechenden Entscheidungsbaum.
 - Jedes Sortierverfahren muss jede mögliche Permutation erzeugen können.
 - Der Entscheidungsbaum hat somit $l = n!$ Blätter.
 - Die Höhe des Entscheidungsbaums ist eine untere Schranke für die Anzahl der Vergleiche:

$$h \geq \log_2 l = \log_2 n! = \Omega(N \log N)$$

Gleichung 3.19
Cormen S. 58

3.5 Counting-, Radix- und Bucket sort

- Gibt es Sortierverfahren, die nicht ‚vergleichend‘ sind?
 - Im allgemeinen nicht, da wir für die zu sortierenden Objekte lediglich die Ordnungsrelation kennen.
- **Annahme 1:** Die zu sortierenden Schlüssel sind ganzzahlig, paarweise verschieden und aus $\{1, \dots, N\}$
 - TRIVIAL_SORT (A)
for $i = 1$ **to** N
 $B[A[i]] = A[i]$
return B
- **Annahme 2:** Die zu sortierenden Schlüssel sind ganzzahlig aus $\{0, \dots, k\}$, $k = O(N)$
 - Idee: Zähle die Anzahl der Elemente mit kleinerem Schlüssel.
 - → Countingsort

Countingsort

- Feld A: Eingabe, Feld B: sortierte Ausgabe
- Feld C[0...k]: Zähler-Array, ändert die Bedeutung während des Alg.
- COUNTING-SORT (A, B, k)

```
1 for( i = 0 to k ) C[i] = 0
2 for( j = 1 to A.length ) C[A[j]] = C[A[j]] + 1
    // C[i] enthält die Anzahl der Elemente, deren Schlüssel gleich i sind
3 for( i = 1 to k ) C[i] = C[i] + C[i-1]
    // C[i] enthält die Anzahl der Elemente, deren Schlüssel  $\leq$  i sind
4 for j = A.length downto 1
5     B[C[A[j]]] = A[j]
6     C[A[j]] = C[A[j]] - 1
7 return B
```

- Laufzeit von Countingsort: $T(N,k) = O(N + k)$
Für $k = O(N)$ sortiert Countingsort in Linearzeit.

Countingsort: Ein Beispiel

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

(a)

	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

(c)

	1	2	3	4	5	6	7	8
B		0					3	

	0	1	2	3	4	5
C	1	2	4	6	7	8

(d)

	1	2	3	4	5	6	7	8
B		0				3	3	

	0	1	2	3	4	5
C	1	2	4	5	7	8

(e)

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

(f)

■ Beispiel für N=8, k=5

- a) nach Zeile 2 b) nach Zeile 3 c) nach 1. Iteration Zeile 4
- d) nach 2. Iteration e) nach 3. Iteration f) nach Zeile 7

Radixsort (Sortieren durch Fächer verteilen)

- **Definition:** Ein Sortierverfahren heißt *stabil*, genau dann wenn für alle $i < j$, $i, j \in \{1, \dots, N\}$ mit $A[i] = A[j]$ gilt $\pi(i) < \pi(j)$
- **Satz:** Countingsort ist ein stabiles Sortierverfahren.
- **Annahme 3:** Sortierung erfolgt nach einem m -adischen Schlüssel $A[i] = (k_1, k_2, \dots, k_d)$; $\forall 1 \leq j \leq d$: k_j ist ganzzahlig und $0 \leq k_j \leq m-1$
- Idee:
 - Verteilphase: Sortiere die Daten in m Fächer nach dem ersten (niederwertigsten!) Schlüssel
 - Sammelphase: Sammle die Daten wieder zu einer Liste
 - wiederhole mit dem nächsten Schlüssel
 - wichtig: im i -ten Durchlauf bleibt die relative Ordnung des $i-1$ ten Durchlaufs erhalten, d.h. die Verteilphase setzt ein stabiles Sortierverfahren als Subroutine voraus

Radixsort (Sortieren durch Fächer verteilen)

- Beispiel: N= 7, m= 10, d= 3

329	720	720	329
457	355	329	355
657	436	436	436
839	839
436	657	355	657
720	329	457	720
355	839	657	839

- RADIX-SORT (A, d, m)

```
for j = 1 to d
    // key[j]: Attribut, beschreibt den j-ten Schlüssel von A
    COUNTING-SORT( A.key[j], B, m-1 )
    A = B
```

Radixsort mit Warteschlangen

```
■ RADIXSORT(A, d, m)
  // L: array [0...m-1] von Warteschlangen
  for( j = 0 to m-1 ) L[j] = init()
  for t = 1 to d           // sortiere nach Schlüssel t
    // Verteilphase
    for( i = 1 to N ) ENQUEUE(L[A[i].key[t]], A[i] )
    // Sammelphase
    i = 1
  for j = 0 to m-1
    while not EMPTY(L[j])
      A[i] = DEQUEUE(L[j])
      i = i+1
```

Radixsort: Laufzeit- und Speicherbedarf

■ Parameter

- m: Anzahl Schlüsselwerte
- d: Anzahl Schlüsselelemente
- N: Anzahl der Datensätze

■ Laufzeit $T(N,d,m) = O(d(N + m))$

Fall 1: $d = O(1)$, $m = O(N)$: $O(N)$

Fall 2: Alle Schlüssel verschieden: $d \geq \log_m N$: $O(N \log N)$

Bucketsort

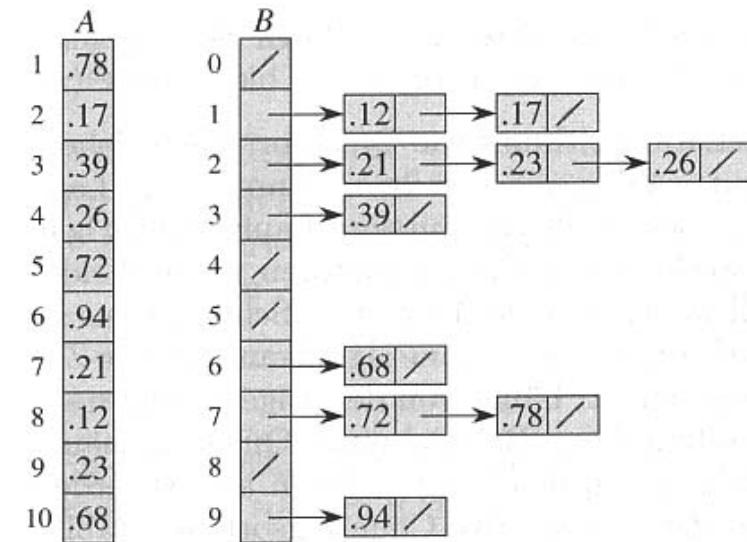
■ **Annahme 4:** Für alle Element $A[i]$ gilt: $A[i] \in \mathbb{R}$, $0 \leq A[i] < 1$. Die Elemente sind in $[0,1)$ gleichverteilt.

■ Idee:

- Teile den Wertebereich in N gleich große Teilintervalle
 $T[i] = [i/N, (i+1)/N)$, für $0 \leq i < N$
- Erzeuge N Fächer $B[0, \dots, N-1]$ mit N linearen Listen
- Sortiere durch Fächerverteilung und verbinde die Listen

■ BUCKET-SORT (A)

```
1 for i = 1 to A.length
2   LIST-INSERT(B[ $\lfloor A.length * A[i] \rfloor$ ], A[i])
3 i = 1
4 for j = 0 to A.length-1
5   INSERTION-SORT(B[j])
6   while not EMPTY( B[j] )
7     A[i] = LIST-HEAD[B[j]];
8     B[j] = LIST-TAIL[B[j]];
9     i = i+1
```



Bucketsort: Laufzeitanalyse

- Bucketsort ohne Zeile 5 benötigt $O(N)$ Zeit. Sei $n_i = \text{length}[B[i]]$:
INSERTION-SORT für Liste $B[i]$ benötigt $O(n_i^2)$ Zeit.

- Worst-Case: $n_0 = N$, $n_i = 0$ für alle $i > 0$: $T_{\text{Bucket}}(N) = O(N^2)$
Dieses Szenario widerspricht allerdings der Gleichverteilungsannahme.
- Worst-Case erwartete Laufzeit:

$$T_{\text{bucket}}(N) = cN + \sum_{i=0}^{N-1} cn_i^2$$

$$E[T_{\text{bucket}}(N)] = E\left[cN + \sum_{i=0}^{N-1} cn_i^2\right] = cN + \sum_{i=0}^{N-1} cE[n_i^2]$$

- Angenommen, es gilt: $E[n_i^2] = 2 - 1/N$

dann folgt: $E[T_{\text{bucket}}(N)] = cN + \sum_{i=0}^{N-1} c(2 - 1/N) = cN(3 - 1/N) = O(N)$

```
1 for i = 1 to A.length
2   LIST-INSERT(B[└A.length*A[i]┘], A[i])
3 i = 1
4 for j = 0 to A.length-1
5   INSERTION-SORT(B[j])
6 while not EMPTY( B[j] )
7   A[i] = LIST-HEAD[B[j]];
8   B[j] = LIST-TAIL[B[j]];
9   i = i+1
```

Bucketsort: Laufzeitanalyse

■ Bleibt zu zeigen: $E[n_i^2] = 2 - 1/N$

Sei X_{ij} das Ereignis, dass $A[j]$ in Bucket i eingesortiert wird. Dann gilt:

$$E[n_i^2] = E\left[\left(\sum_{j=1}^N X_{ij}\right)^2\right] = \sum_{j=1}^N E[X_{ij}^2] + \sum_{j=1}^N \sum_{\substack{k=1 \\ k \neq j}}^N E[X_{ij} X_{ik}]$$

(Ausmultiplizieren des Quadrats zur Trennung stochastisch abhängiger von unabhängigen Termen)

Ereignis X_{ij} tritt mit Wahrscheinlichkeit $1/N$ auf (Gleichverteilung!):

$$E[X_{ij}^2] = 1^2 \frac{1}{N} + 0^2 \left(1 - \frac{1}{N}\right) = \frac{1}{N}$$

Ereignisse X_{ij} und X_{ik} sind unabhängig: $E[X_{ij} X_{ik}] = E[X_{ij}]E[X_{ik}] = \frac{1}{N^2}$

Insgesamt folgt: $E[n_i^2] = \sum_{j=1}^N \frac{1}{N} + \sum_{j=1}^N \sum_{\substack{k=1 \\ j \neq k}}^N \frac{1}{N^2} = N \frac{1}{N} + N(N-1) \frac{1}{N^2} = 2 - \frac{1}{N}$

Bucketsort: Laufzeitanalyse

■ Alternative Herleitung für: $E[n_i^2] = 2 - 1/N$

Bestimmung über die Varianz: $E[n_i^2] = V[n_i] + E^2[n_i]$

(dadurch entfällt die explizite Berechnung des Funktionsquadrats)

Im folgenden laufen alle Summen ohne expl. Indizes von $j=1, \dots, N$.

Es gilt:

$$E[n_i] = E\left[\sum X_{ij}\right] = \sum E[X_{ij}] = \sum \frac{1}{n} = 1$$

$$\begin{aligned} V[X_{ij}] &= \sum_{z=0}^1 \Pr\{X_{ij} = z\} (z - E[X_{ij}])^2 \\ &= \underbrace{\left(1 - \frac{1}{N}\right)}_{z=0} \underbrace{\left(0 - \frac{1}{N}\right)^2}_{z=1} + \underbrace{\left(\frac{1}{N}\right)}_{z=1} \underbrace{\left(1 - \frac{1}{N}\right)^2}_{z=0} = \left(1 - \frac{1}{N}\right) \left(\left(\frac{1}{N}\right)^2 + \frac{1}{N} - \left(\frac{1}{N}\right)^2 \right) = \frac{1}{N} - \frac{1}{N^2} \end{aligned}$$

$$V[n_i] = V\left[\sum X_{ij}\right] = \sum V[X_{ij}] = \sum \left(\frac{1}{N} - \frac{1}{N^2}\right) = 1 - \frac{1}{N}$$

$$E[n_i^2] = V[n_i] + E^2[n_i] = 1 - \frac{1}{N} + 1^2 = 2 - \frac{1}{N}$$

Achtung: Die Umformung setzt stochastische Unabhängigkeit der X_{ij} voraus.

Abschließende Kommentare zu Sortieren in Linearzeit

- Wenn wir nur die Vergleichsrelation für die Sortierung von Objekten zur Verfügung haben, gilt die **untere Laufzeitschranke** $\Omega(N \log N)$.
- Wenn arithmetische Operationen auf Schlüsseln möglich sind, kann diese Schranke unter Umständen durchbrochen werden:
 - Annahme 1: Die zu sortierenden Schlüssel sind ganzzahlig, paarweise verschieden und aus $\{1, \dots, N\}$ → **Trivalsort**
 - Annahme 2: Die zu sortierenden Schlüssel sind ganzzahlig aus $\{0, \dots, k\}$, $k = O(N)$ → **Countingsort**
 - Annahme 3: Sortierung erfolgt nach einem m -adischen Schlüssel $A[i] = (k_1, k_2, \dots, k_d)$; $\forall 1 \leq j \leq d: k_j$ ist ganzzahlig und $0 \leq k_j \leq m-1$ → **Radixsort**
 - Annahme 4: Für alle Element $A[i]$ gilt: $A[i] \in \mathbb{R}$, $0 \leq A[i] < 1$. Die Elemente sind in $[0, 1)$ gleichverteilt. → **Bucketsort**
 - **Bucketsort** kann auch verwendet werden, wenn wir die Verteilung der Elemente kennen und eine Funktion entwickeln können, die die Elemente auf $[0, \dots, N-1]$ gleichverteilt.

3.6 Algorithmen für Auswahlprobleme

■ Suchproblem:

- Eingabe: eine sortierte Folge A mit N Schlüsseln $A[i]$ und einen Suchschlüssel q
- Ausgabe: Index eines Elements k mit $A[k] = q$ oder -1, falls $A[k] \neq q$ für alle k

■ Auswahlproblem:

- Eingabe: Eine Menge A aus N (paarweise verschiedenen) Zahlen und einen Suchindex i mit $1 \leq i \leq N$
- Ausgabe: Das Element $x \in A$ mit $A[k] < x$ für i-1 Elemente $A[k]$

Suchen in sortierten Folgen

■ Algorithmus 1: Lineare Suche

```
■ LINEAR-SEARCH(A, q)
  for( i = 0 to A.length ) if( A[i] == q ) return i
  return -1
```

■ Laufzeit: $O(k)$, falls q in der Folge vorkommt, $O(N)$ sonst

■ Algorithmus 2: Binäre Suche (Aufruf mit BINARY-SEARCH(A,1,A.length,q))

```
■ BINARY-SEARCH(A, l, r, q)
  if( l > r ) return -1
  else
    m ← ⌊(l+r)/2⌋
    if( A[m] > q ) BINARY-SEARCH(A, l, m-1, q)
    elseif( A[m] < q ) BINARY-SEARCH(A, m+1, r, q)
    else return m
```

■ Laufzeit: $T(N) = T(N/2) + c = O(\log N)$

Suchen in sortierten Folgen

■ Algorithmus 3: Exponentielle Suche

■ Idee: Verdopple den rechten Rand bis $A[r] > q$ gilt, führe binäre Suche im Intervall $[r/2, \dots, r]$ durch

■ EXPONENTIAL-SEARCH (A, q)

1 $r \leftarrow 1$

2 **while** ($A[r] < q$ **and** $r < N$) $r = \min(2r, N)$

3 **return** BINARY-SEARCH ($A, r/2+1, r, q$)

■ Laufzeit:

◆ Fall 1: $\exists k : A[k] = q$

Schleife 2: $\log_2 k + 1$ Durchläufe, dann gilt $r/2 < k \leq r$ $O(\log k)$

Binäre Suche: $O(\log r/2) = O(\log r - 1) =$ $O(\log k)$

Gesamtzeit: $O(\log k)$

◆ Fall 2: $\forall k : A[k] \neq q$ $O(\log N)$

Die Laufzeit von EXPONENTIAL-SEARCH ist **output-sensitiv**, d.h. sie hängt vom Resultat der Suche ab.

Das Auswahlproblem

■ Auswahlproblem:

- Eingabe: Eine Menge A aus N (paarweise verschiedenen) Zahlen und einen Suchindex i mit $1 \leq i \leq N$
- Ausgabe: Das Element $x \in A$ mit $A[k] < x$ für $i-1$ Elemente $A[k]$

■ Bsp:

- $i=1$: Minimum-Problem $O(N)$
- $i=N$: Maximum-Problem $O(N)$
- $i=N/2$: Median-Problem ?

■ Algorithmus 1: Iteratives Löschen der Minima

```
TRIVIAL-SELECT (A, i)
    n ← A.length
    while i > 0
        k = MIN-INDEX (A, n ) // gibt den Index des min. Elements aus A[1...n]
        SWAP (A[k], A[n])
        i = i-1; n = n-1;
    return A[n+1]
```

Laufzeit: $T(N,i) = \Theta(i N) = O(N^2)$

Der QUICK-SELECT Algorithmus

■ Algorithmus 2: Sortieren

```
SORT-SELECT(A, i)
```

```
HEAPSORT(A)
```

```
output A[i]
```

Laufzeit: $O(N \log N)$

■ Algorithmus 3: Partielle Sortierung mit Quicksort

```
RAND-SELECT( A, l, r, i )  
1 if( l == r ) return A[l]      // Annahme: l ≤ i ≤ r  
2 else  
3   q = RAND-PARTITION(A, l, r)  
4   if( i == q ) return A[q]  
5   elseif( i < q ) RAND-SELECT(A, l, q-1, i )  
6   else RAND-SELECT(A, q+1, r, i)
```

Achtung: RAND-SELECT() und RANDOMIZED-SELECT() [Cormen Kap. 9.2] weichen geringfügig voneinander ab, lösen aber das gleiche Problem.

Der RAND-SELECT Algorithmus

■ Beispiel: N=8, i=6

	index: 1 2 3 4 5 6 7 8							
	A: 3 8 14 2 9 23 1 6							
nach 3.:	A: 3 1 2 6 9 23 8 14 \leq \geq							
	A: x x x x 9 23 8 14							
nach 3.:	A: x x x x 9 8 14 23 \leq \geq							
	A: x x x x 9 8 x x							
nach 3.:	A: x x x x 8 9 x x q \geq							
	A: x x x x x 9 x x l=r=i							

Analyse des RAND-SELECT Algorithmus

- Rekurrenzgleichung für Rand-Select:
 - Sei q die Position des Pivot-Elements, dann gilt
$$T(N) = T(q - 1) + a \text{ N} \text{ falls } i < q, \quad T(N) = T(N - q) + a \text{ N} \text{ sonst.}$$
 - Worst-Case: $q=N \Rightarrow T(N) = T(N-1) + a \text{ N} = \Theta(N^2)$
- Wie hoch ist die erwartete Laufzeit $E[T(N)]$ für Rand-Select?
 - Alle Elemente $A[k]$ können gleichwahrscheinlich als Pivot-Element gewählt werden.
 - Im Worst-Case liegt der gesuchte Rang i immer in der größeren Partition

$$\begin{aligned} E[T(N)] &\leq \sum_{k=1}^N \Pr\{q = k\} E[T(\max(k-1, N-k) + aN)] \\ &= \sum_{k=1}^N \frac{1}{N} E[T(\max(k-1, N-k) + aN)] = \frac{1}{N} \sum_{k=1}^N E[T(\max(k-1, N-k))] + aN \end{aligned}$$

Analyse des RAND-SELECT Algorithmus

- Wie hoch ist die erwartete Laufzeit $E[T(N)]$ für Rand-Select?

- Für $k < N/2$ ergibt sich für k und $N-k+1$ der gleiche Summand:

$$\max(k-1, N-k) = N-k = \max((N-k+1)-1, N-(N-k+1))$$

$$E[T(N)] \leq \frac{1}{N} \sum_{k=1}^N E[T(\max(k-1, N-k))] + aN \leq \frac{2}{N} \sum_{k=\lfloor N/2 \rfloor}^{N-1} E[T(k)] + aN$$

- Welche Laufzeitschranke sollten wir intuitiv erwarten?

- Im Vergleich zu Quicksort erfolgt nur ein rekursiver Aufruf, also höchstens $O(N \log N)$
 - Angenommen, $k=N/2$, dann folgt $T(N)=T(N/2)+cN = O(N)$

Analyse des RAND-SELECT Algorithmus

- Annahme: $E[T(N)] = O(N)$, d.h. $E[T(N)] \leq c N$ für ein konstantes c
- Beweis:

$$\begin{aligned} E[T(N)] &\leq \frac{2}{N} \sum_{k=\lfloor N/2 \rfloor}^{N-1} ck + aN \\ &= \frac{2c}{N} \sum_{k=0}^{\lceil N/2 \rceil - 1} (\lfloor N/2 \rfloor + k) + aN \\ &= \frac{2c(\lceil N/2 \rceil)}{N} \lfloor N/2 \rfloor + \frac{2c}{N} \frac{(\lceil N/2 \rceil - 1)(\lceil N/2 \rceil)}{2} + aN \\ &\leq \frac{2c(N/2 + 1)(N/2)}{N} + \frac{2c}{N} \frac{(N/2)(N/2 + 1)}{2} + aN \\ &= \frac{cN}{2} + c + \frac{cN}{4} + \frac{c}{2} + aN = \frac{3cN}{4} + \frac{3c}{2} + aN \\ &= cN - \left(\frac{cN}{4} - \frac{3c}{2} - aN \right) \leq cN \quad \text{für } N \geq \frac{6c}{c - 4a} \end{aligned}$$

Analyse des RAND-SELECT Algorithmus (Cormen-Variante)

$$\begin{aligned} E[T(N)] &\leq \frac{2}{N} \sum_{k=\lfloor N/2 \rfloor}^{N-1} ck + aN \\ &= \frac{2c}{N} \left(\sum_{k=1}^{N-1} k - \sum_{k=1}^{\lfloor N/2 \rfloor - 1} k \right) + aN \\ &= \frac{2c}{N} \left(\frac{(N-1)N}{2} - \frac{(\lfloor N/2 \rfloor - 1)\lfloor N/2 \rfloor}{2} \right) + aN \\ &\leq \frac{2c}{N} \left(\frac{(N-1)N}{2} - \frac{(N/2-2)(N/2-1)}{2} \right) + aN \\ &= \frac{2c}{N} \left(\frac{3N^2}{4} + \frac{N}{2} - 2 \right) + aN = \frac{3cN}{4} + \frac{c}{2} + aN \\ &= cN - \left(\frac{cN}{4} - \frac{c}{2} - aN \right) \leq cN \quad \text{für } N \geq \frac{2c}{c-4a} \end{aligned}$$

Der SELECT-Algorithmus

- Gibt es einen deterministischen Algorithmus für das Auswahlproblem mit asymptotisch linearer Laufzeit im Worst-Case?
- Idee: Verwende Alternative zu Rand-Select und garantiere, dass die Position q des Pivotelements nahe bei $N/2$ liegt.

```
■ SELECT( A, l, r, i )  
1 if( l == r ) return A[l]      // Annahme: l ≤ i ≤ r  
2 else  
3   q = FIND-PIVOT-POSITION( A, l, r )  
4   SWAP( A[q], A[r] )  
5   q = PARTITION(A, l, r)  
6   if( i == q ) return A[q]  
7   elseif( i < q ) SELECT(A, l, q-1, i )  
8   else SELECT(A, q+1, r, i)
```

Der SELECT-Algorithmus

- Idee: Teile Menge in Fünfer-Gruppen, berechne die Fünfer-Mediane und den Median der Fünfer-Mediane:

- FIND-PIVOT-POSITION (A, l, r)

```
n = r-l+1
```

```
// berechne die Mediane von je 5 benachbarten Elementen
```

```
for i = 1 to  $\lceil n/5 \rceil$ 
```

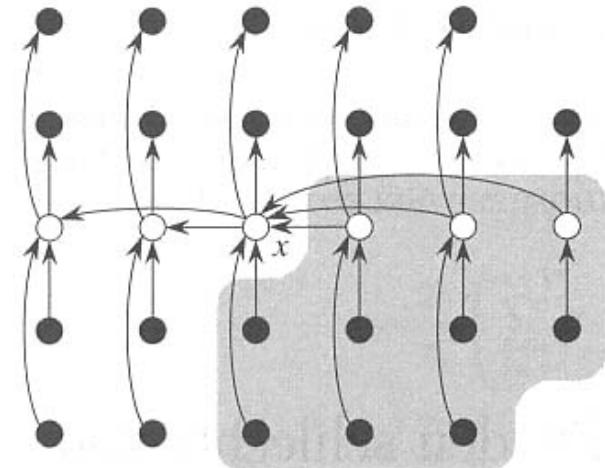
```
    INSERTION-SORT(  $A[ (l + 5(i-1)), \dots, \min( l+5i, r ) ]$  )
```

```
    M[i] = A[  $\min( l + 5i - 3, r )$  ]
```

```
// berechne rekursiv den Median der Mediane-von-5
```

```
x = SELECT( M, 1,  $\lceil n/5 \rceil$ ,  $\lfloor \lceil n/5 \rceil / 2 \rfloor$  )
```

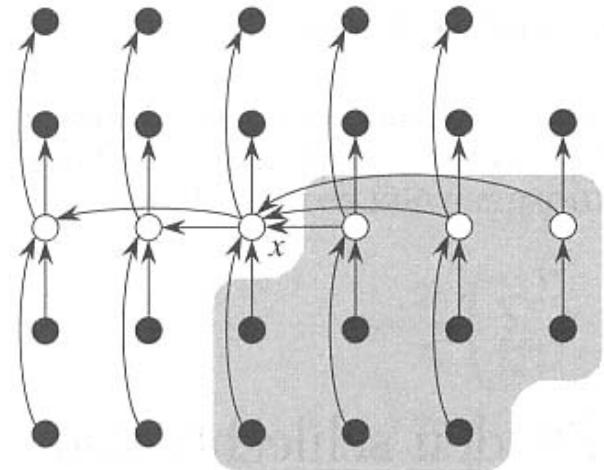
```
return (  $\min( l + 5 x - 3, r )$  )
```



Die Analyse des SELECT-Algorithmus

- FIND-PIVOT-POSITION: Wie viele Elemente aus $A[l, \dots, r]$ sind größer als $A[x]$?
 - die Hälfte der Fünfer-Mediane
 - + je zwei weitere Elemente
 - ohne die letzte Gruppe und
 - ohne die Gruppe, die x enthält

$$3\left(\left\lceil \frac{1}{2} \left\lceil \frac{N}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3N}{10} - 6 = G(N)$$



- Der rekursive Aufruf von SELECT (Zeile 7 oder 8) erfolgt mit maximal $7N/10 + 6$ Elementen.

Die Analyse des SELECT-Algorithmus

■ Die Rekurrenz-Gleichung für SELECT:

$$T(N) = \begin{cases} c & \text{falls } n \leq 70 \\ T(\lceil N/5 \rceil) + T(7N/10 + 6) + aN & \text{sonst} \end{cases}$$

SELECT-Aufruf in FIND-PIVOT-POSITION SELECT-Aufruf in SELECT - Bestimmung der Fünfer-Mediane - Aufruf von PARTITION

- Annahme: $T(N) = O(N)$, d.h. $T(N) \leq cN$ für ein konstantes c
- Beweis: Für alle $N \leq 70$ ist $T(N) = c$ konstant. Sei $N > 70$, dann gilt:

$$\begin{aligned} T(N) &\leq c \left\lceil \frac{N}{5} \right\rceil + c \left(\frac{7N}{10} + 6 \right) + aN \\ &\leq \frac{cN}{5} + c + \frac{7cN}{10} + 6c + aN = \frac{9cN}{10} + 7c + aN \\ &= cN - \left(\frac{1}{10}cN - 7c - aN \right) \leq cN \\ \Leftrightarrow \frac{1}{10}cN - 7c - aN &\geq 0 \Leftrightarrow c \geq 10a \left(\frac{N}{N-70} \right) \quad \text{für } N > 70 \end{aligned}$$

Abschließende Bemerkungen zum Auswahlproblem

- Das Auswahlproblem kann in asymptotisch erwarteter linearer Zeit mit dem **RAND-SELECT-Algorithmus** gelöst werden.
- Es gibt einen deterministischen Algorithmus (**SELECT-Algorithmus**), der das Auswahlproblem im Worst Case in asymptotisch linearer Zeit löst.
- Der **SELECT-Algorithmus** ist aufgrund hoher Konstanten nur von Interesse, falls die Eingaben sehr groß sind und eine Laufzeitgarantie gefordert ist.
- Lehren für die Algorithmenentwicklung:
 - Ein randomisierter, rekursiver Algorithmus, der in asymptotisch linearer Zeit die Eingabe gleichverteilt um einen Faktor $\alpha \in [0,1)$ verkleinert, hat asymptotisch eine erwartete lineare Laufzeit.
 - Ein deterministischer, rekursiver Algorithmus, der in asymptotisch linearer Zeit die Eingabe um einen Faktor $\alpha \in [0,1)$ verkleinert, hat asymptotisch eine lineare Laufzeit.

Instant Feedback

- Bitte gern die Vorlesung heute bewerten:
- <https://feedback.informatik.uni-hamburg.de/AD/wise2019-2020>



optionaler Kommentar