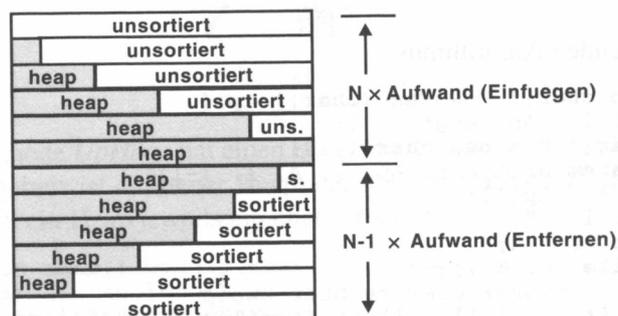




Heapsort



- Beispiel für einen eleganten Algorithmus, der auf einer effizienten Datenstruktur (dem Heap) beruht [Williams, 1964]
- Daten liegen in einem Array der Länge n vor
- 1. Erstelle aus dem gegebenen Array einen Heap (**DownHeap**)
- 2. Tausche erstes und letztes Element des Arrays
 - dann ist das größte Element an der letzten Position – wo es hingehört
 - es bleiben $n-1$ Elemente, die an die entsprechende Position müssen
 - das Array von $n-1$ Elementen ist jedoch kein Heap mehr
 - verschiebe das (neue) Element der Wurzel in einen ihrer Unterbäume, damit das Array wieder ein Heap wird (**DownHeap**)
 - wiederhole Schritt 2 bis das Array sortiert ist
- Trick: verwende Array selbst zur Speicherung des Heaps





Erstellung eines Heaps

- benutze **DownHeap** um ein Array *A* in einen Heap umzuwandeln
- rufe **DownHeap** für jedes Element nach der Bottom-Up-Methode auf

```
BuildHeap(A)  
for i in range( len(A)/2-1, -1, -1 ):  
    DownHeap( A, i, len(A) )
```

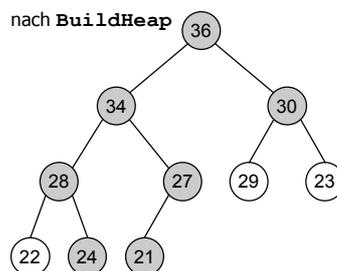
```
DownHeap(A, l, r)  
# A = array  
# A[l..r-1] = Bereich, der "heap-ifiziert" werden soll  
# A[l] = Wurzel, die "versickert" werden soll
```



Beispiel

Eingabe-Array

| | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|
| 24 | 21 | 23 | 22 | 36 | 29 | 30 | 34 | 28 | 27 |
|----|----|----|----|----|----|----|----|----|----|





Korrektheit von **BuildHeap**



- **Schleifeninvariante:** zu Beginn jeder Iteration der for-Schleife ist jeder Knoten $i+1, i+2, \dots, n-1$ die Wurzel eines Heaps
- **Initialisierung:**
 - vor der ersten Iteration ist $i = \lfloor n/2 \rfloor$
 - Knoten $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \dots, n-1$ sind Blätter und daher Wurzeln von Heaps
- **Erhaltung der Invariante:**
 - durch die Schleifeninvariante sind die Kinder des Knotens i Heaps
 - daher macht **DownHeap(i)** aus Knoten i eine Heap-Wurzel (die Heap-Eigenschaft von höher nummerierten Knoten bleibt erhalten)
 - Verminderung von i stellt die Schleifen-Invariante für die nächste Iteration wieder her.



Laufzeit von **BuildHeap**



- lockere obere Schranke (*loose upper bound*):
 - Kosten von einem **DownHeap**-Aufruf \times Anzahl von **DownHeap**-Aufrufen $\rightarrow O(\log(n)) \cdot O(n) = O(n \log(n))$
- engere Schranke (*tighter upper bound*):
 - Kosten für einen Aufruf von **DownHeap** an einem Knoten hängen von seiner Höhe h ab $\rightarrow O(h)$
 - Knotenhöhe h liegt zwischen 0 und $\lfloor \log(n) \rfloor$ (hier: Blätter = Höhe 0!)
 - Anzahl der Knoten mit Höhe h ist $\left\lceil \frac{n}{2^{h+1}} \right\rceil$

- festere Schranke für $T(\text{BuildHeap})$:

$$\begin{aligned}
 T(\text{BuildHeap}) &\in \sum_{h=1}^{\lfloor \log(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \\
 &= O\left(n \underbrace{\sum_{h=1}^{\lfloor \log(n) \rfloor} \frac{h}{2^h}}_{\leq \sum_{h=0}^{\infty} \frac{h}{2^h} = 2} \right) \\
 &= O(n)
 \end{aligned}$$

→ Erstellt einen Heap von einem unsortierten Array in linearer Zeit!

HeapSort (A)

```

HeapSort(A)
BuildHeap(A)
for i in range(len(A)-1, -1, -1):
    A[0] <-> A[i]
    DownHeap(A, 0, i)
  
```

- BuildHeap benötigt $O(n)$ und jeder der $n-1$ Aufrufe von **DownHeap** benötigt $O(\log(n))$
- daher gilt $T(n) \in O(n + (n-1) \cdot \log(n)) = O(n \log(n))$



State-of-the-Art für Heapsort-Verfahren



- HEAPSORT (Floyd 1964):

$$C_{\max}(n) = 2n \log n + O(n)$$

- BOTTOM-UP-HEAPSORT (Wegener 1993):

$$C_{\max}(n) = 1,5n \log n + O(n)$$

- WEAK-HEAPSORT (Dutton 1993):

$$C_{\max}(n) = n \log n + 0.1n$$

- RELAXED-HEAPSORT:

$$C_{\max}(n) = n \log n - 0.9n$$



Laufzeitvergleiche



- Laufzeiten der schnellen Sortieralgorithmen

- zufällige Daten (Laufzeit in Sekunden)
- Java, 450 MHz Pentium II

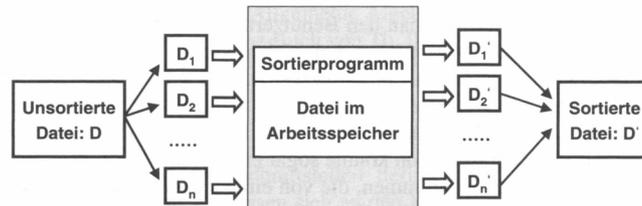
| | N=1000000 | N=2000000 | N=3000000 | N=4000000 |
|------------------|-----------|-----------|-----------|-----------|
| ShellSort | 2,5 | 5,8 | 9,5 | 13,2 |
| QuickSort | 0,8 | 1,7 | 2,6 | 3,5 |
| MergeSort | 2,0 | 4,1 | 6,2 | 8,6 |
| HeapSort | 2,5 | 5,6 | 9,1 | 12,7 |
| DistributionSort | 0,6 | 1,2 | 1,8 | 2,5 |



Externes Sortieren



- Was macht man, wenn die Daten nicht alle auf einmal in den Speicher passen?
 - Teile die große, externe Datei D in n Teile D_1, \dots, D_n , die jeweils im Speicher intern sortiert werden können
 - die jeweils sortierten Dateien D_1', \dots, D_n' werden anschließend zu der insgesamt sortierten Datei D' zusammengemischt
- meist Variante von Mergesort

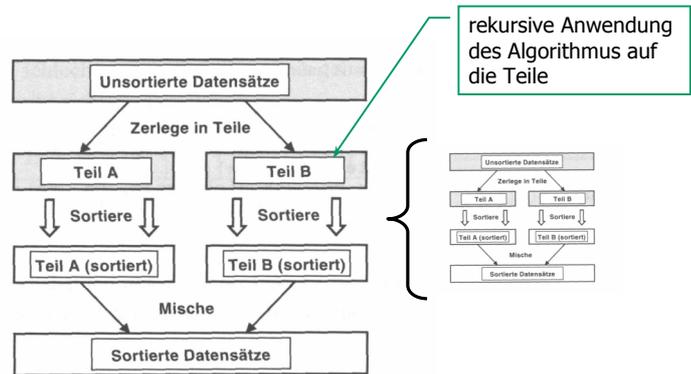


Mergesort

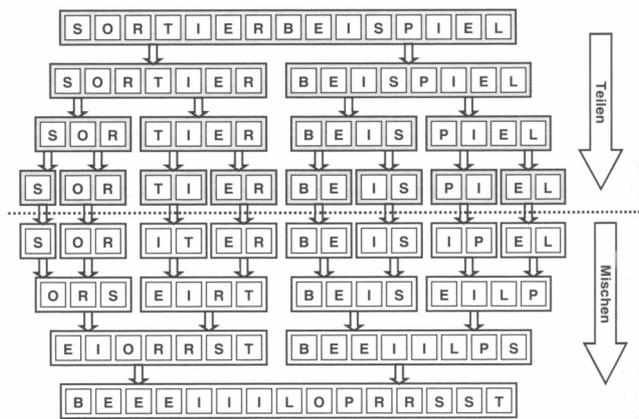


- Idee:
 - teile die ursprüngliche Menge an Datensätzen in zwei Hälften
 - sortiere die beiden Teilmengen
 - mische die beiden sortierten Hälften wieder zusammen (engl. *merge*)
 - wähle dazu das kleinere der beiden Elemente, die an der jeweils ersten Stelle der beiden Datensätze stehen
 - wende das Verfahren rekursiv auf die beiden Hälften an, um diese zu sortieren

Prinzip



Beispiel



```

def mergesort( A ):
    return rek_mergesort( A, 0, len(A)-1 )

def rek_mergesort( A, lo, hi ):
    if hi <= lo:
        return
    mid = (lo + hi) / 2
    A1 = rek_mergesort( A, lo, mid )
    A2 = rek_mergesort( A, mid+1, hi )
    return merge( A1, A2 )

```

```

def merge(a, b):
    result = []
    if len(a) == 0: return b
    if len(b) == 0: return a

    i = j = 0
    while i < len(a) and j < len(b):
        if a[i] <= b[j]:
            result.append( a[i] )
            i += 1
        else:
            result.append( b[j] )
            j += 1

    while i < len(a):
        result.append( a[i] )
    while j < len(b):
        result.append( b[j] )

    return result

```



Eigenschaften



- Algorithmus ist sehr übersichtlich und einfach
- Optimierung:
 - Anlegen von Hilfsarrays kostet Zeit
 - besser ein großes Hilfsarray anlegen und immer wieder benutzen
- *In-place* Sortierung (also ohne Hilfsarray) (aka *in situ*) möglich, aber sehr kompliziert
- Aufwand:
 - $N \cdot \log(N)$
 - $\log(N)$ viele Etagen, Aufwand pro Etage proportional N , gilt **auch im worst case**
- sonst nicht besonders schnell, da viel umkopiert wird



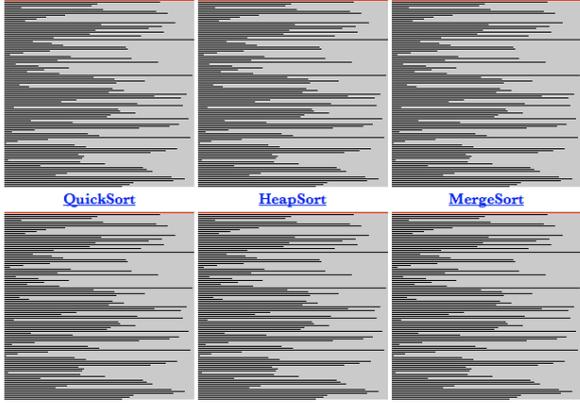
- Vorteile:
 - Besser geeignet, wenn sequentieller Zugriff schnell, und "random" Zugriff langsam (z.B.: Listen, Bänder, langsames RAM aber schneller Cache)
 - Leichter parallelisierbar
 - **Stabiler** Sortier-Algo



Algorithmus-Animationen

Animated Sorting Algorithms

BubbleSort SelectionSort InsertionSort



QuickSort HeapSort MergeSort

Click on the pictures to start the algorithms

<http://www.inf.ethz.ch/~staerk/algorithms/SortAnimation.html>

G. Zachmann Informatik 2 - SS 06 Sortieren 86

Untere Schranke für allgemeine Sortierverfahren

- Viele Verfahren bis jetzt, manche mit $O(n^2)$ manche mit $O(n \log n)$
- Prinzipielle Frage: wie schnell können wir überhaupt werden?
- **Satz:**
Zum Sortieren einer Folge von n Schlüsseln mit einem allgemeinen Sortierverfahren sind im Worst-Case ebenso wie im Average-Case mindestens $\Omega(n \log n)$ Vergleichsoperationen zwischen zwei Schlüsseln erforderlich.
- Beweis durch Modellierung von allgemeinen Sortierverfahren als **Entscheidungsbäume**

G. Zachmann Informatik 2 - SS 06 Sortieren 87



Wichtiges Charakteristikum von allgemeinem Sortieren



- **Allgemeines Sortieren = Vergleichsbasiertes Sortieren:**
 - Nur Vergleich von Elementpaaren wird benutzt, um die Ordnung einer Folge zu erhalten
 - Für alle Algos gilt: pro Vergleich eine konstante Anzahl weitere Operationen (z.B. 2 Elemente kopieren, Schleifenzähler erhöhen, ...)
 - Daher: untere Schranke der Vergleichszahl = untere Schranke für die Komplexität eines vergleichsbasiertes Sortieralgorithmus'
- Alle bisher behandelten Sortierverfahren sind vergleichsbasiert
- Die bisher beste **Worst-Case-Komplexität** ist $O(n \log n)$ (Mergesort, Heapsort)
- Voriger Satz besagt: worst-case Komplexität von Merge- und Heapsort ist optimal (ebenso average-case von Quicksort)



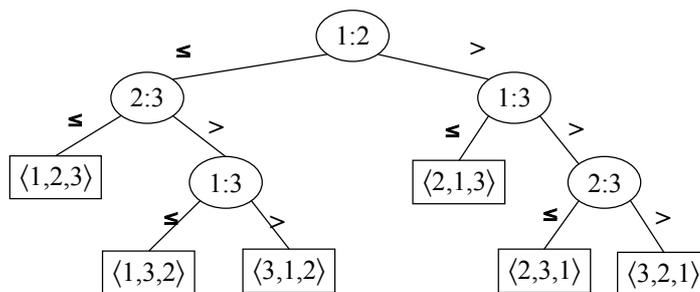
Entscheidungsbaum



- Abstraktion eines Sortierverfahrens durch einen **Binärbaum**
- **Entscheidungsbaum** stellt Folge von Vergleichen dar
 - von irgendeinem Sortieralgorithmus
 - für Eingaben einer vorgegebenen Größe
- lässt alles andere (Kontrollfluß und Datenverschiebungen) außer Acht, es werden nur Vergleiche betrachtet
- **interne Knoten** bekommen Bezeichnung $i:j$ = die Positionen der Elemente im Eingangsfeld, die verglichen werden
- **Blätter** werden mit **Permutationen** $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$ bezeichnet, die der Algorithmus bestimmt

Beispiel

- Entscheidungsbaum für Insertionsort mit drei Elementen



- beinhaltet $3! = 6$ Blätter

- Ausführen des Sortieralgorithmus' für bestimmte Eingabe entspricht dem **Verfolgen eines Weges** von der Wurzel zu einem Blatt
- Entscheidungsbaum bildet alle möglichen Ausführungsabläufe ab
- Bei jedem internen Knoten findet ein Vergleich $a_i \leq a_j$ statt.
 - für $a_i \leq a_j$, folge dem linken Unterbaum
 - sonst, folge dem rechten Unterbaum
- An einem Blatt ist Ordnung $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$ festgelegt.
- Ein korrekter Sortieralgorithmus *muß* alle Permutationen erzeugen können
 - M.a.W.: jede der $n!$ Permutationen muß bei mindestens einem Blatt des Entscheidungsbaumes vorkommen



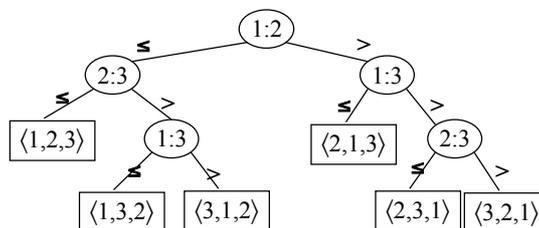
Untere Schranke für Worst-Case

- Anzahl der Vergleiche im Worst-Case eines Sortieralgorithmus'
 - = Länge des längsten Weges im Entscheidungsbaum von der Wurzel zu irgendeinem Blatt
 - = die Höhe des Entscheidungsbaumes
- untere Schranke für die Laufzeit = untere Schranke für die Höhe aller Entscheidungsbäume, in denen jede Permutation als erreichbares Blatt vorkommt



Beispiel: Optimales Sortierverfahren für drei Elemente

- Der Entscheidungsbaum für das Sortierproblem auf drei Elementen hat
 - 6 Blätter
 - 5 interne Knoten
 - Unabhängig von Anordnung der Knoten: es muß einen Worst-Case-Weg mit einer Länge ≥ 3 geben.





- **Satz:** Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche im Worst-Case.

- **Beweis:**

- Es reicht, die Höhe eines Entscheidungsbaumes zu bestimmen.
- h = Höhe, l = Anzahl der Blätter im Entscheidungsbaum
- Im Entscheidungsbaum für n Elemente gilt: $l \geq n!$
- Im Binärbaum mit der Höhe h gilt: $l \leq 2^h$

- also: $n! \leq l \leq 2^h \Rightarrow n! \leq 2^h \Rightarrow h \geq \log(n!)$

- Stirling Approximation: $n! > \left(\frac{n}{e}\right)^n$

- somit : $h \geq \log(n!) \geq \log\left(\left(\frac{n}{e}\right)^n\right)$
 $= n \log(n) - n \log(e)$
 $= \Omega(n \log(n))$



Untere Schranke für Average-Case

- **Satz:**
Jedes vergleichsbasierte Sortierverfahren benötigt $\Omega(n \log(n))$ Vergleiche im Mittel (Average-Case)

- Wir beweisen zunächst ...

- **Lemma:**

Die mittlere Tiefe eines Blattes eines Binärbaumes mit k Blättern ist mindestens $\log_2(k)$.

- Beweis durch Widerspruch

- Annahme: Lemma ist falsch
- Sei T der **kleinste** Binärbaum, der Lemma verletzt; T habe k Blätter



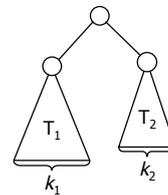
Beweis von Lemma



- $k \geq 2$ muss gelten (Lemma gilt ja für $k = 1$)
- T hat linken Teilbaum T_1 mit k_1 Blättern und rechten Teilbaum T_2 mit k_2 Blättern
 - es gilt $k_1 + k_2 = k$
 - bezeichne mit $\bar{d}(T)$ die mittlere Tiefe von Baum T
 - da $k_1, k_2 < k$ sind, gilt Lemma für T_1, T_2 :

$$\bar{d}(T_1) \geq \log(k_1)$$

$$\bar{d}(T_2) \geq \log(k_2)$$



- für jedes Blatt von T gilt: Tiefe dieses Blattes, bezogen auf die Wurzel von $T =$ Tiefe + 1, bezogen auf die Wurzel von T_1 bzw. T_2

$$k\bar{d}(T) = \sum_{\text{Blätter } b \text{ in } T} \text{Tiefe von } b$$

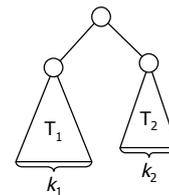
- also:

$$\text{Summe aller Blattiefen in } T = k_1(\bar{d}(T_1) + 1) + k_2(\bar{d}(T_2) + 1)$$

$$\Rightarrow \bar{d}(T) = \frac{1}{k} (k_1(\bar{d}(T_1) + 1) + k_2(\bar{d}(T_2) + 1))$$

$$\geq \frac{k_1}{k} (\log(k_1) + 1) + \frac{k_2}{k} (\log(k_2) + 1)$$

$$= \frac{1}{k} (k_1 \log(2k_1) + k_2 \log(2k_2)) =: f(k_1, k_2)$$





- Funktion $f(k_1, k_2)$ nimmt, unter der Nebenbedingung $k_1 + k_2 = k$, Minimum bei $k_1 = k_2 = k/2$ an

- also

$$\bar{d}(\mathcal{T}) \geq \frac{1}{k} \left(\frac{k}{2} \log(k) + \frac{k}{2} \log(k) \right) = \log(k)$$

- Widerspruch zur Annahme!



Beweis des Satzes

- Mittlere Laufzeit eines Sortierverfahrens = mittlere Tiefe eines Blattes im Entscheidungsbaum
- Entscheidungsbaum hat $k \geq N!$ viele Blätter
- also

$$\bar{d} \geq \log N! \geq \log \left(\frac{N}{2} \right)^{\frac{N}{2}} = \frac{N}{2} \log \left(\frac{N}{2} \right)$$

$$\bar{d} \in \Omega(N \log(N))$$



Lineare Sortierverfahren



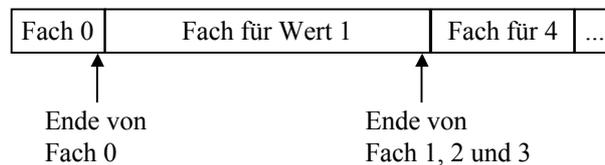
- Bisherige Sortieralgorithmen basieren auf den Operationen
 - **Vergleich** zweier Elemente
 - **Vertauschen** der Elemente
- führt bestenfalls zum Aufwand $N \cdot \log(N)$ [schneller geht es nicht]
- **Distributionsort**: Klasse von Sortierverfahren, die zusätzliche Operationen (neben Vergleichen) verwenden, z.B. arithmetische Operationen, Zählen, Zahldarstellung als Ziffernfolge, ...
 - Allgemeines Schema (ganz grob):
 - Verteilung der Daten auf Fächer (*distribute*)
 - Einsammeln der Daten aus den Fächern, wobei **Ordnung innerhalb der Fächer erhalten bleiben muß(!)** (*gather*)



Counting-Sort



- Vorbedingung: Keys kommen aus einem **diskreten** Bereich
- Zunächst simple Idee: reserviere für jeden mögl. Wert ein Fach
 - Problem: jedes Fach müsste potentiell Platz für alle Datensätze bieten
- Trick: verwende nur **ein** Ausgabearray B und mache die Fächer genau so groß, wie sie benötigt werden. Dazu muß man sich in einem zweiten Array C die Fächergrenzen merken:





Algorithmus



- Annahme: Es gibt einen, dem Algorithmus Countingsort bekannten Parameter k , so daß für die Eingabefolge (a_1, \dots, a_n) gilt: $\forall i : 0 \leq a_i \leq k$
- Algorithmusidee:
 - für alle $i, 0 \leq i \leq k$, bestimme Anzahl C_i der a_j mit $a_j \leq i$:

$$C_i := |\{a_j \in A \mid a_j \leq i\}|, \quad C_{-1} = 0$$

$$C_i - C_{i-1} = |\{a_j \in A \mid a_j = i\}|$$
 - erzeuge Array B , genauso groß wie A
 - kopiere a_j mit $a_j = i$ in Felder

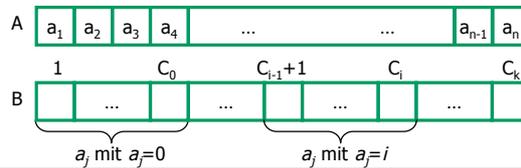


Illustration von Countingsort



```

C = [0] * (k+1)
for j in range( 0, len(A) ):
    C[A[j]] += 1
# C[i] = # elements a_j = i
for i in range( 1, k+1 ):
    C[i] += C[i-1]
# C[i] = # elements a_j <= i
for j in range( len(A), 0 ):
    C[A[j]] -= 1
    B[ C[A[j]] ] = A[j]

```

A

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 2 | 5 | 3 | 0 | 2 | 3 | 0 | 3 |

C

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 2 | 2 | 4 | 7 | 7 | 8 |

C

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 2 | 0 | 2 | 3 | 0 | 1 |

Illustration von Countingsort

```

C = [0] * (k+1)
for j in range( 0, len(A) ):
    C[A[j]] += 1
# C[i] = # elements a_j = i
for i in range( 1, k+1 ):
    C[i] += C[i-1]
# C[i] = # elements a_j <= i
for j in range( len(A), 0 ):
    C[A[j]] -= 1
    B[ C[A[j]] ] = A[j]

```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| A | 2 | 5 | 3 | 0 | 2 | 3 | 0 | 3 |

| | | | | | | |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| C | 0 | 2 | 2 | 4 | 7 | 7 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| B | 0 | 0 | 2 | 2 | 3 | 3 | 3 | 5 |

G. Zachmann Informatik 2 - SS 06 Sortieren 104

Analyse

```

C = [0] * (k+1)
for j in range( 0, len(A) ):
    C[A[j]] += 1
# C[i] = # elements a_j = i
for i in range( 1, k+1 ):
    C[i] += C[i-1]
# C[i] = # elements a_j <= i
for j in range( len(A), 0 ):
    C[A[j]] -= 1
    B[ C[A[j]] ] = A[j]

```

} $O(k)$

} $O(n)$

} $O(k)$

} $O(n)$

- **Satz:** Counting-Sort besitzt Laufzeit $O(n+k)$
- **Korollar:** Gilt $k \in O(n)$, so besitzt Counting-Sort Laufzeit $O(n)$

G. Zachmann Informatik 2 - SS 06 Sortieren 105