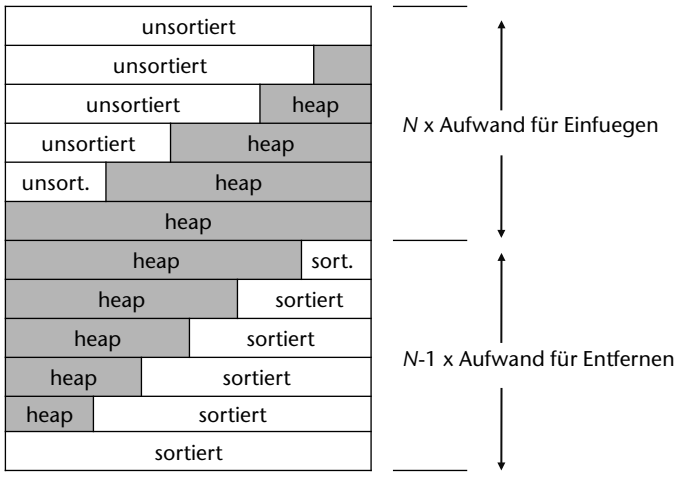
Heapsort

- 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 Max-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
   - Stelle Heap-Eigenschaft wieder her (**DownHeap**)
   - Wiederhole Schritt 2 bis das Array sortiert ist

- Trick: verwende das Array selbst zur Speicherung des Heaps

- Prinzipielle Vorgehensweise:



Erstellung eines Heaps

- Rufe **DownHeap** für jedes Element nach der Bottom-Up-Methode auf:

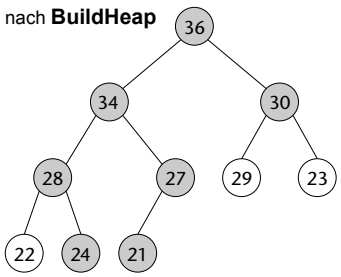
```
BuildHeap(A):
for i = n/2-1, ..., 0:
  DownHeap( A, i, n-1 )
```

```
DownHeap(A, l, r):
# A = array
# A[l…r] = Bereich, der "heap-ifiziert" werden soll
# A[l] = Wurzel, die "versickert" werden soll
# Precondition: die beiden Kinder von A[l] sind
# korrekte Heaps
```


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, …, 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, … n-1 sind Blätter und daher (trivialerweise) 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**

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

- Engere Schranke (Fortsetzung):

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

$$= O\left(n \underbrace{\sum_{h=1}^{\lfloor \log(n) \rfloor} \frac{h}{2^h}}_{\leq \sum_{h=0}^{\infty} \frac{h}{2^h} = 2} \right)$$

$$= O(n)$$

- Fazit: man kann einen Heap aus einem unsortierten Array in linearer Zeit erstellen!

Der Algorithmus für Heapsort

```
HeapSort(A):

BuildMaxHeap(A)
for i = n-1, …, 1:
    swap A[0] and A[i]
    DownHeap( A, 0, i-1 )
```

- BuildHeap benötigt $O(n)$ und jeder der n-1 Aufrufe von **DownHeap** benötigt $O(\log n)$
- Daher gilt:

$$T(n) \in O\big(n + (n-1)\log n\big) = O\big(n \log n\big)$$

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):

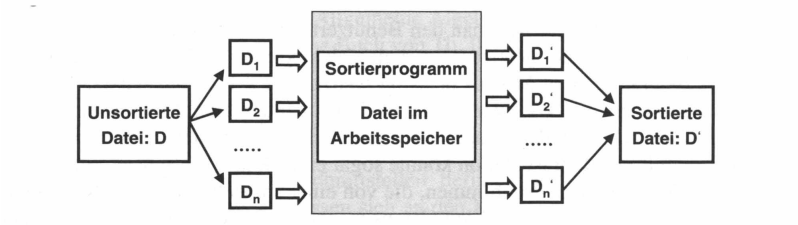
$$C_{\max}(n) = n \log n + 0.1n$$

- RELAXED-HEAPSORT:

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

Externes Sortieren

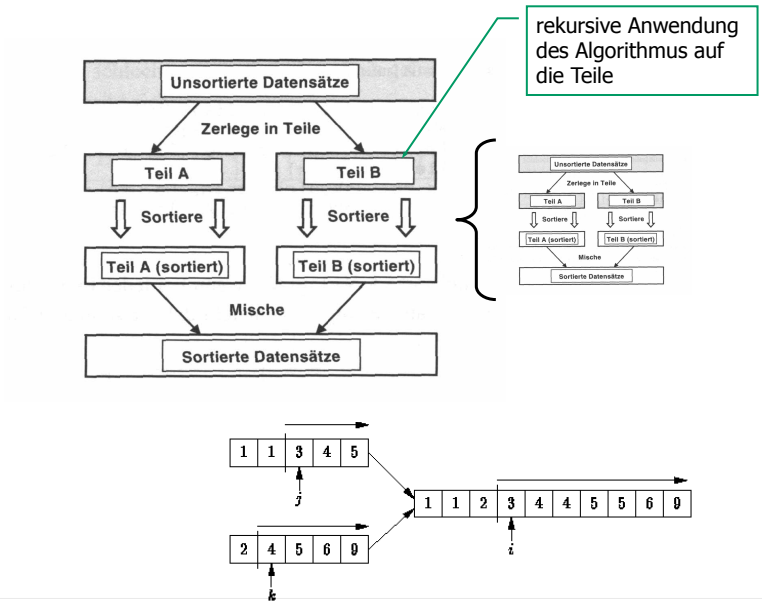
- Was macht man, wenn die Daten nicht alle auf einmal in den Speicher passen?
  - Teile die große, externe Datei D in *n* Teile D_1, ..., D_n, die jeweils im Speicher intern sortiert werden können
  - Die jeweils sortierten Dateien D_1', ..., D_n' werden anschließend zu der insgesamt sortierten Datei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



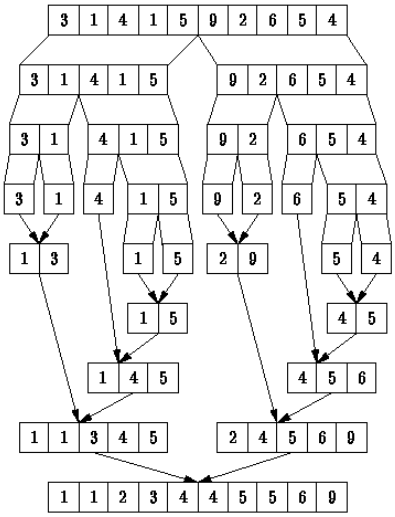
Mergesort

- 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

Das Prinzip



- Beispiel



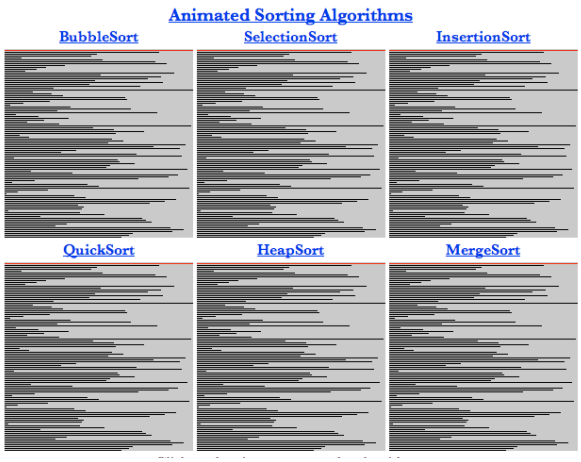

```
def mergesort( A ):
    return rek_mergesort( A, 0, len(A)-1 )


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):
    if len(a) == 0:  return b
    if len(b) == 0:  return a

    result = []
    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 zu implementieren
- Aufwand: N*log(N)
  - log(N) viele Etagen, Aufwand pro Etage in O(N), gilt auch im worst case
  - Nicht besonders schnell, da viel umkopiert wird
- Optimierung:
  - Ständiges Anlegen von Hilfsarrays kostet Zeit
  - Besser ein großes Hilfsarray anlegen und immer wieder benutzen
- *In-place* Sortierung (ohne Hilfsarray) möglich, aber sehr kompliziert
- Vorteile:
  - Besser geeignet, wenn sequentieller Zugriff schnell, und "random" Zugriff langsam (z.B.: Listen, Bänder, Festplatten)
  - Stabiler Sortier-Algo

Algorithmus-Animationen



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