Erster Sortier-Algorithmus: Bubblesort

- Die Idee des Algo:
  - Vergleiche von links nach rechts jeweils zwei Nachbar-elemente und vertausche deren Inhalt, falls sie in der falschen Reihenfolge stehen;
  - Wiederhole dies, bis alle Elemente richtig sortiert sind;
  - Analogie: die kleinsten Elemente steigen wie Luftblasen zu ihrer richtigen Position auf (je nachdem, ob man aufsteigend oder absteigend sortiert)

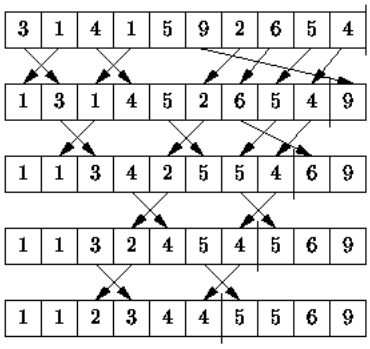
Effiziente Python-Implementierung

```
def bubblesort( a ):
    for k in ...:
        for i in range( 0, len(a)-1 ):
            if a[i] > a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
```

```
def bubblesort( a ):
    for k in range( 0, len(a)-1 ):
        for i in range( 0, len(a)-1 ):
            if a[i] > a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
```

```
def bubblesort( a ):
    for k in range( len(a)-1, 0, -1 ):
        for i in range(0,k):
            if a[i] > a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
```

- Beispiel:



Korrektheitsbeweis

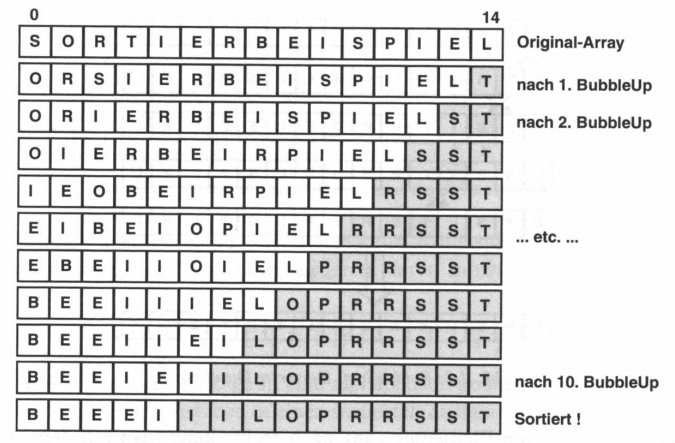
- Schleifeninvariante:
  - Nach dem 1. Durchlauf befindet sich das größte Element an der richtigen Stelle
  - Nach dem 2. Durchlauf auch das 2.-größte, etc.
  - Nach dem *i*-ten Durchlauf befinden sich die *i* größten Elemente an der richtigen Position (und damit in der richtigen Reihenfolge)
- Nach spätestens *N*-1 Durchgängen ist das Array sortiert
- Da bei jedem Durchlauf auch andere Elemente ihre Position verbessern, ist häufig der Vorgang bereits nach weniger als *N*-1 Durchgängen beendet

- Kleine Optimierung: Test auf vorzeitiges Ende

```
def bubblesort( a ):
    for k in range( len(a)-1, 0, -1 ):
        sorted = true
        for i in range (0,k):
            if a[i] > a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
                sorted = false
        if sorted:
            break
```

- Für die Laufzeit-Analyse: was ist der Worst-Case?
  - Wenn das kleinste Element ganz hinten (oben) steht

- Beispiel



Aufwand von Bubblesort

- Laufzeitberechnung für den *worst case*:

```
def bubblesort( a ):
    k = len(a)-1
    while k >= 0:
        for i in range (0,k):
            if a[i]>a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
```

O(1) O(k) T(n)

$$T(n) \in \sum_{k=1}^{n} O(k) = O(\sum_{k=1}^{n} k) = O(\frac{1}{2}n(n+1)) = O(n^2)$$

- Für den *best case* (für den Code mit "early exit"): $T(n) \in O(n)$
  - Beweis: Übungsaufgabe
- Im *average case* (o.Bew.): $T(n) \in O(n^2)$

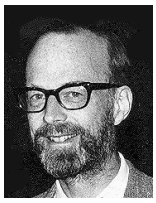
Weitere "einfache" Sortierverfahren

- Insertion Sort, Selection Sort, u.a.

http://www.youtube.com/watch?v=INHF_5RIxTE (und auf der VL-Homepage)

Quicksort

- C.A.R. Hoare, britischer Informatiker, erfand 1960 Quicksort
- Bis dahin dachte man, man müsse die einfachen Sortieralgorithmen durch raffinierte Assembler-Programmierung beschleunigen

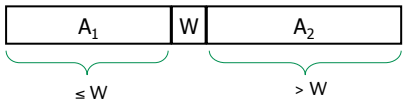


C. A. R. Hoare

- Quicksort zeigt, daß es sinnvoller ist, nach besseren Algorithmen zu suchen
- Einer der schnellsten bekannten allgemeinen Sortierverfahren
- Idee:
  - Vorgegebenes Sortierproblem in kleinere Teilprobleme zerlegen
  - Teilprobleme rekursiv sortieren
  - Allgemeines Algorithmen-Prinzip: *divide and conquer* (divide et impera)

Der Algorithmus

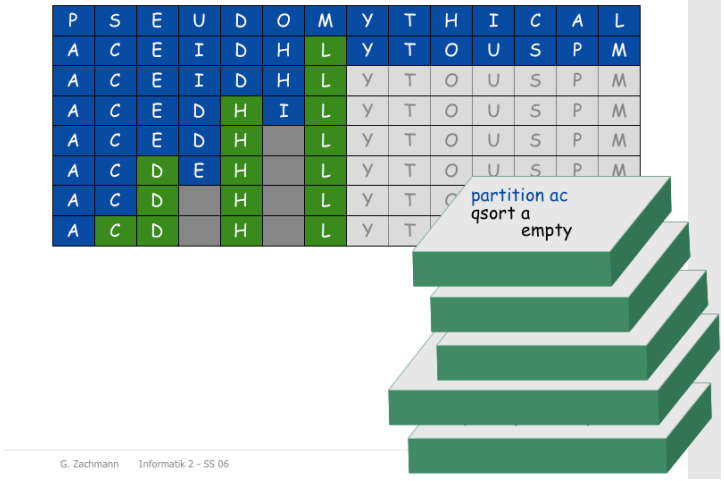
1. Wähle irgend einen Wert W des Arrays A
2. Konstruiere Partitionierung des Arrays mit folgenden Eigenschaften:



   - A_1 und A_2 sind noch unsortiert!
3. Wenn man jetzt A_1 und A_2 sortiert, ist das Problem gelöst
4. A_1 und A_2 sortiert man natürlich wieder mit ... Quicksort

Algo-Animation

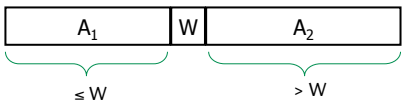
Quicksort



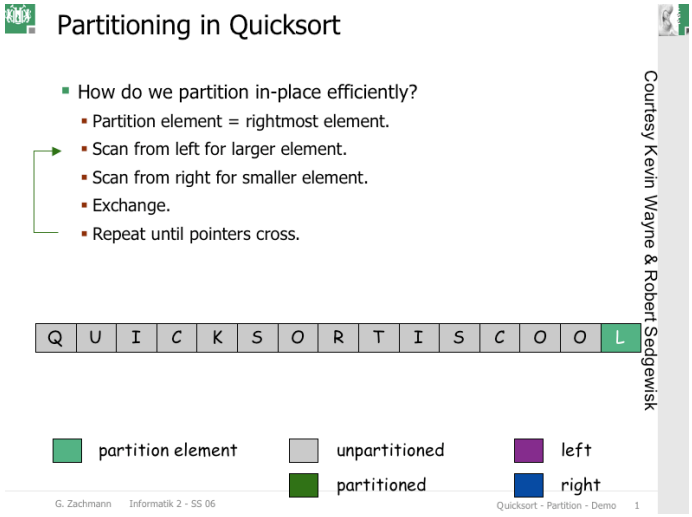
- Konstruktion der Partition ist die eigentliche Kunst / Arbeit bei Quicksort!

1. Wähle ein Element W im Array (dieses heißt Pivot-Element)
2. Suche ein i von links mit A[i]>W
3. Suche ein j von rechts mit A[j]≤W
4. Vertausche A[i] und A[j]
5. Wiederhole bis i ≥ j-1 gilt
6. Speichere W "dazwischen"

- Resultat:



Algo-Animation



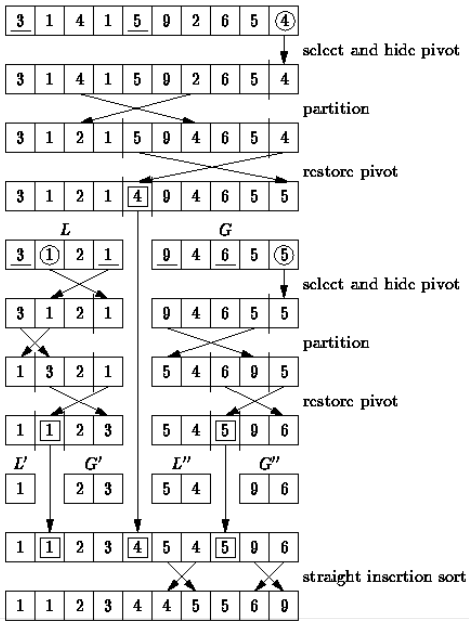
Python-Implementierung

```
def quicksort( A ):                          # wrapper
   recQuicksort( A, 0, len(A)-1 )

def recQuicksort( A, links, rechts ):
    if rechts <= links :
        return                               # base case

    # find pivot and partition array in-place
    pivot = partition( A, links, rechts )

    # sort smaller array slices
    recQuicksort( A, links, pivot-1 )
    recQuicksort( A, pivot+1, rechts )
```

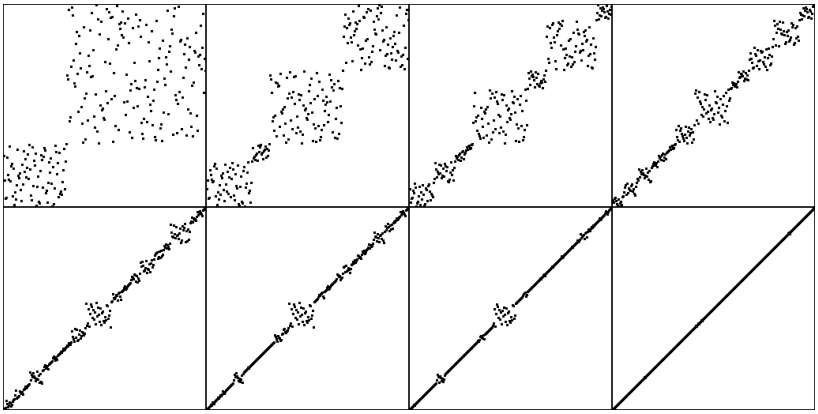
```
def partition( A, links, rechts ):
    pivot = rechts          # choose right-most as pivot
    i, j = links, rechts-1

    while i < j:            # quit when i,j "cross over"
        # find elem > pivot from left
        while A[i] <= A[pivot] and i < rechts:
            i += 1
        # find elem < pivot from right
        while A[j] > A[pivot] and j > links:
            j -= 1

        if i < j:
            # swap mis-placed elements
            A[i], A[j] = A[j], A[i]

    # put pivot at its right place and return its pos
    A[i], A[pivot] = A[pivot], A[i]
    return i
```

Diese Implementierung enthält noch einen Bug! Wer findet ihn?

Beispiel-Durchlauf von Quicksort



Visualisierung von Quicksort



LO = nach der ersten Iteration, RU = fertig.
X-Achse = Index, Y-Achse = Wert in diesem Array-Element

Korrektheit der Partitionierung

- Ann.: wähle das letzte Element A_r im Teil-Array $A_{l..r}$ als Pivot
- Bei der Partitionierung wird das Array in vier Abschnitte, die auch leer sein können, eingeteilt:
  1. $A_{l..i-1}$ → Einträge dieses Abschnitts sind ≤ pivot
  2. $A_{j+1..r-1}$ → Einträge dieses Abschnitts sind > pivot
  3. A_r = pivot
  4. $A_{i..j}$ → Status bzgl. *pivot* ist unbekannt
- Dies ist eine Schleifeninvariante

- Initialisierung: vor der ersten Iteration gilt:
  - $A_{l..i-1}$ und $A_{j+1..r-1}$ sind leer – Bedingungen 1 und 2 sind (trivial) erfüllt
  - r ist der Index des Pivots – Bedingung 3 ist erfüllt

```
i, j = l, r-1
p = A[r]
while i < j:
    # find elem > pivot from left
    while A[i] <= p and i < r:
        i += 1
    # find elem < pivot from right
    while A[j] > p and j > l:
        j -= 1
    # swap mis-placed elems
    if i < j:
        A[i], A[j] = A[j], A[i]
[…]
```


- Erhaltung der Invariante (am Ende des Schleifenrumpfes):
  - Nach erster **while**-Schleife gilt: **A[i] > p** oder **i=r**
  - Nach zweiter **while**-Schleife gilt: **A[j] ≤ p** oder **j=l**
  - Vor **if** gilt: falls **i<j**, dann ist **A[i] > p ≥ A[j]**
    - was dann durch den **if**-Body "repariert" wird
  - Nach **if** gilt wieder Schleifeinvariante

```
i, j = l, r-1
p = A[r]
while i < j:
    # find elem > pivot from left
    while A[i] <= p and i < r:
        i += 1
    # find elem < pivot from right
    while A[j] > p and j > l:
        j -= 1
    # swap mis-placed elems
    if i < j:
        A[i], A[j] = A[j], A[i]
[…]
```

- Beendigung:
  - Nach der while-Schleife gilt:

$$i \geq j \ \wedge \ (A_i > A_r \vee i = r)$$

  - D.h.:
    - $A_{l..i-1} \leq pivot$
    - $A_{i+1..r-1} > pivot$
    - $A_r = pivot$
    - der vierte Bereich , $A_{i..j}$, ist leer
- Die letzte Zeile vertauscht A_i und A_r:
  - *Pivot* wird vom Ende des Feldes zwischen die beiden Teil-Arrays geschoben
  - damit hat man $A_{l..i} \leq pivot$ und $A_{i+1..r} > pivot$
- Also wird die Partitionierung korrekt ausgeführt

```
i, j = l, r-1
p = A[r]
while i < j:
    […]
A[i], A[r] = A[r], A[i]
return i
```

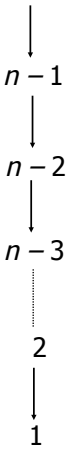

Laufzeit des Algorithmus'

- Die Laufzeit von Quicksort hängt davon ab, ob die Partitionen ausgeglichen sind oder nicht
- Der Worst-Case:
  - Tritt auf, wenn jeder Aufruf zu am wenigsten ausgewogenen Partitionen führt
  - Eine Partitionen ist am wenigsten ausgewogen, wenn
    - das Unterproblem 1 die Größe n–1 und das Unterproblem 2 die Größe 0, oder umgekehrt, hat
    - *pivot* ≥ alle Elemente $A_{l..r-1}$ oder *pivot* < alle Elemente $A_{l..r-1}$
  - Also: jeder Aufruf ist am wenigsten ausgewogen, wenn
    - das Array sortiert oder umgekehrt sortiert ist

- Laufzeit für Worst-Case-Partitionen bei jedem Rekursionsschritt:

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \text{PartitionTime}(n) \\
&= T(n-1) + \Theta(n) \\
&= \sum_{k=1}^{n} \Theta(k) \;=\; \Theta\left(\sum_{k=1}^{n} k\right) \\
&\in \Theta(n^2)
\end{aligned}
$$

Rekursionsbaum für Worst-Case-Partitionen



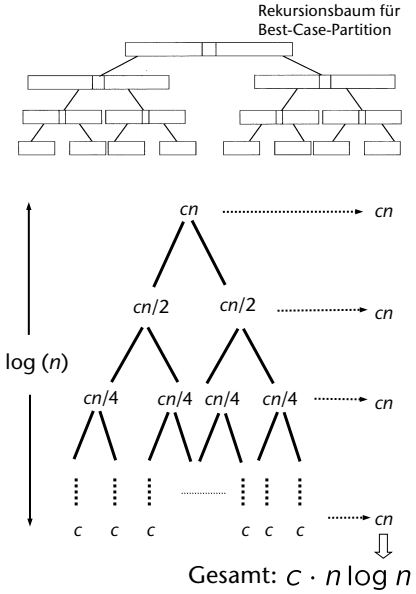
Laufzeit bei Best-Case-Partitionierung

- Größe jedes Unterproblems $\leq \frac{n}{2}$ genauer: ein Unterproblem hat die Größe $\lfloor \frac{n}{2} \rfloor$, das andere die Größe $\lceil \frac{n}{2} \rceil - 1$
- Laufzeit:

$$
\begin{aligned}
T(n) &\leq 2T(\tfrac{n}{2}) \\
&\quad + \text{PartitionTime}(n) \\
&= 2T(\tfrac{n}{2}) + cn
\end{aligned}
$$

- Ann.: $T(1) = c$
- Also: $T(n) \in \Theta(n \log(n))$

Rekursionsbaum für Best-Case-Partition



Gesamt: $c \cdot n \log n$

Auswahl des Pivot-Elementes

- Pivot =
  - "central point or pin on which a mechanism turns", oder
  - "a person or thing that plays a central part or role in an activity"
- Optimal wäre ein Element, das A in zwei genau gleich große Teile partitioniert (Median)
- Exakte Suche macht Laufzeitvorteil von Quicksort wieder kaputt
- Üblich ist: Inspektion von drei Elementen
  - A[li], A[re], A[mid] mit mid=(li+re)/2
  - wähle davon den Median (wertmäßig das mittlere der drei)
  - nennt man dann *"median-of-three quicksort"*
- Alternative: zufälligen Index als Pivot-Element
  - Diese Technik heißt: "Randomisierung"

- Beispiel, wenn man nur A[mid] als Vergleichelement nimmt:

```
SORTIERBEISPIEL
SORTIER B EISPIEL
B ORTIERSEISPIEL
```

  - schlechtest mögliche Partitionierung
- A_2 weiter sortieren:

```
ORTIERSEISPIEL
ORTIER S EISPIEL
ORLIEREEIIP S ST
```

- Beispiel, wenn mittleres Element von A[li], A[re], A[mid] als Pivot-Element verwendet wird:

```
SORTIERBEISPIEL
BEIIIEE L RTSPROS
```

Programm für Median-of-3-Quicksort

```
# Liefert Indizes a,b,c (= Permutation von i,j,k)
# so dass A[a] <= A[b] <= A[c]
def median( A, i, j, k ):
    if A[i] <= A[j]:
        if A[j] <= A[k]:
            return i,j,k
        else:                           # i,k < j
            if A[i] <= A[k]:
                return i,k,j
            else:
                return k,i,j
    else:                               # j < i
        if A[i] <= A[k]:
            return j,i,k
        else:                           # j,k < i
            if A[j] <= A[k]:
                return j,k,i
            else:
                return k,j,i
```



```
def median_pivot( A, links, rechts ):
    middle = (links+rechts) / 2
    l,m,r = median( A, links, middle, rechts )
    A[l], A[m], A[r] = A[links], A[middle], A[rechts]
    return m
```

```
def median_quicksort( A, links, rechts ):
    if rechts <= links :
        return

    # find Pivot and partition array in-place
    pivot = median_pivot( A, links, rechts )
    pivot = partition( A, links+1, pivot, rechts-1  )

    # sort smaller array slices
    median_quicksort( A, links, pivot-1 )
    median_quicksort( A, pivot+1, rechts )
```

State-of-the-Art für Quicksort

- Untere Schranke für average case:
$$C_{av}(n) \geq \lceil \log(n!) \rceil - 1 \approx n \log n - 1,4427n$$
- Ziel: $C_{av}(n) \leq n \log n + cn$ für möglichst kleines c
- Quicksort-Verfahren:
  - QUICKSORT (Hoare 1962)
    $C_{av}(n) \approx 1,386n \log n - 2,846n + O(\log n)$
  - CLEVER-QUICKSORT (Hoare 1962)
    $C_{av}(n) \approx 1,188n \log n - 2,255n + O(\log n)$
  - QUICK-HEAPSORT (Cantone & Cincotti 2000)
    $C_{av}(n) = n \log n + 3n + o(n)$
  - QUICK-WEAK-HEAPSORT
    $C_{av}(n) = n \log n + 0,2n + o(n)$

Der Heap

- Definition Heap :
  ist ein vollständiger Baum mit einer Ordnung ≤, für den gilt, daß jeder Vater ≤ seiner beiden Söhnen ist, d.h.,
$$\forall v : v \leq \text{left}(v) \ \wedge \ v \leq \text{right}(v)$$
- Form:
- Eigenschaft: entlang jedes Pfades von der Wurzel zu einem Knoten sind die Knoten aufsteigend sortiert.

- Spezielle Eigenschaft der Wurzel: kleinstes Element
- Achtung: keine Ordnung zwischen `left(v)` und `right(v)`!
- Obige Definition ist ein sog. "Min-Heap" (analog "Max-Heap")

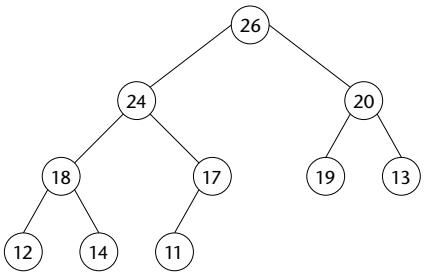
Erinnerung

- Array betrachtet als vollständiger Baum
  - physikalisch – lineares Array
  - logisch – Binärbaum, gefüllt auf allen Stufen (außer der niedrigsten)
- Abbildung von Array-Elementen auf Knoten (und umgekehrt) :
  - Wurzel ↔ $A[1]$
  - links[i] ↔ $A[\,2i\,]$
  - rechts[i] ↔ $A[\,2i+1\,]$
  - Vater[i] ↔ $A[\,\lfloor i/2 \rfloor\,]$

Beispiel

26	24	20	18	17	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10

Max-Heap als Array

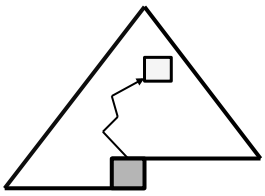


Max-Heap als Binärbaum

- Höhe eines Heaps: $\lfloor \log(n) \rfloor$
- Letzte Zeile wird von links nach rechts aufgefüllt

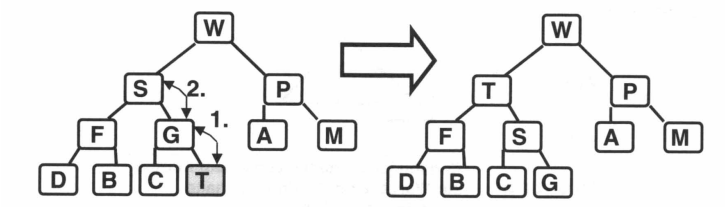
Operationen auf Heaps

- Einfügen eines Knotens:
  - Ex. nur eine mögliche Position, wenn der Baum vollständig bleiben soll
  - Aber im allg. wird dadurch Heap-Eigenschaft verletzt
  - Wiederherstellen mit **UpHeap** (Algorithmus ähnlich zu Bubblesort): vergleiche Sohn und Vater und vertausche gegebenenfalls



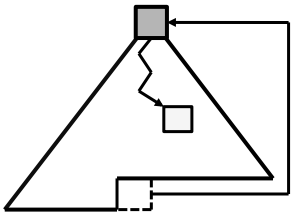
- Bemerkung: ist unabhängig davon, ob Min-Heap oder Max-Heap

- Beispiel:

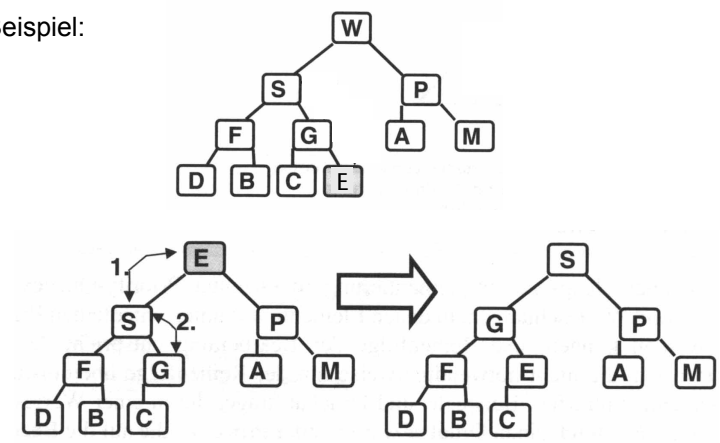


- Aufwand: O(log *n*)

- Löschen der Wurzel:
  - Ersetze Wurzel durch das am weitesten rechts stehende Element der untersten Schicht (Erhaltung der Formeigenschaft des Heaps)
  - Zustand jetzt: beide Teilbäume unter der Wurzel sind immer noch Heaps, aber der gesamte Baum i.A. nicht mehr
  - Wiederherstellen der Ordnungseigenschaft mit **DownHeap**: Vertausche den Vater mit dem kleineren der beiden Söhne (bzw. größeren der beiden für Max-Heap), bis endgültiger Platz gefunden ist



- Beispiel:



- Aufwand: **UpHeap** und **DownHeap** sind beide O(log *n*)

- Heap implementiert eine Verallgemeinerung des FIFO-Prinzips: die Priority-Queue (*p-queue*)
  - Daten werden nur vorne an der Wurzel (höchste Priorität) entfernt (wie bei Queue)
  - Aber Daten werden entsprechend ihres Wertes, der Priorität, einsortiert (nicht hinten)

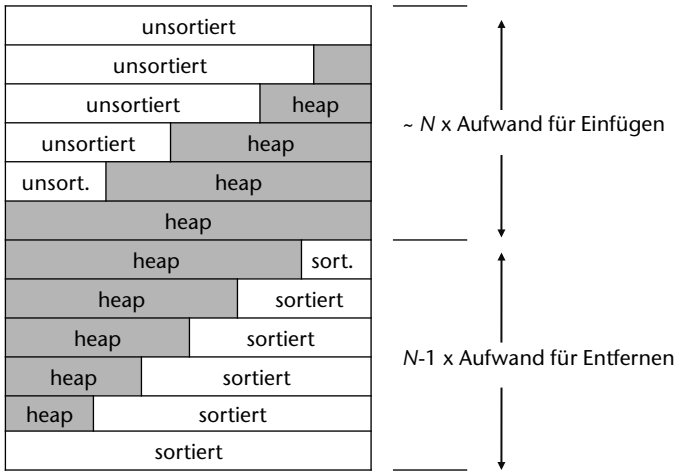
Heapsort

- 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 aus einem Array

- Der Trick:
  - Die Elemente A[n/2] bis A[n-1] bilden schon je einen (trivialen) Heap!
  - Nehmen wir nun elemente n/2-1 dazu, so werden dadurch zwei bestehende Heaps unter eine gemeinsame Wurzel "gehängt" → **DownHeap** anwenden, um für diesen gemeinsam n Heap wieder die Heap-Eigenschaft herzustellen

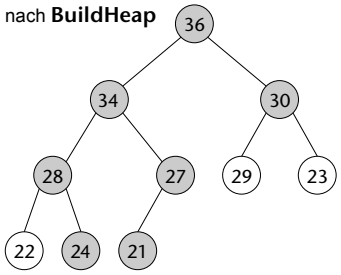
```
BuildHeap(A):
for i = n/2-1, ..., 0:
  DownHeap( A, i, n-1 )
```

```
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):

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

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

Der Algorithmus für Heapsort

```
HeapSort(A):

BuildHeap(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):

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

- BOTTOM-UP-HEAPSORT (Wegener 1993):

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

- WEAK-HEAPSORT (Dutton 1993):

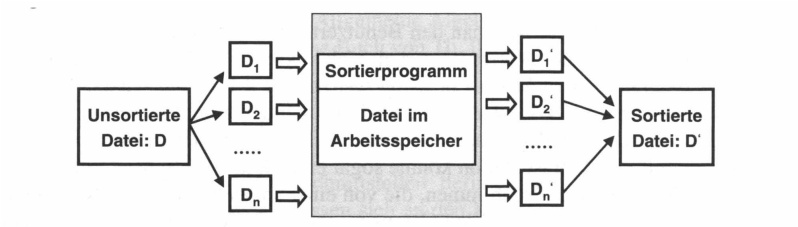
$$C_{\max}(n) = n \log n + 0.1n$$

- RELAXED-HEAPSORT:

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

Externes Sortieren

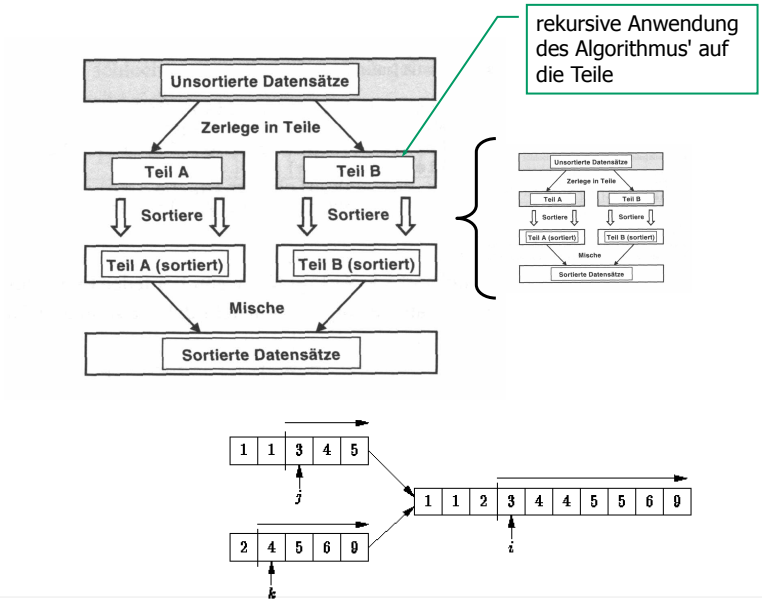
- Was macht man, wenn die Daten nicht alle auf einmal in den Speicher passen?
  - Teile die große, externe Datei D in *n* Teile D_1, ..., D_n, die jeweils im Speicher intern sortiert werden können
  - Die jeweils sortierten Dateien D_1', ..., D_n' werden anschließend zu der insgesamt sortierten Datei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



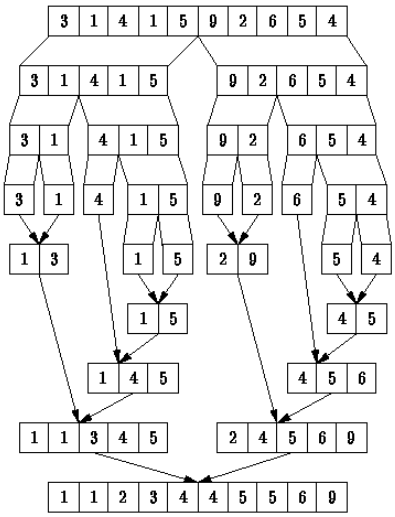
Mergesort

- Idee:
  - Teile die ursprüngliche Menge an Daten 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, l, r ):
    if r <= l:
        return
    mid = (l + r) / 2
    A1 = rek_mergesort( A, l, mid )
    A2 = rek_mergesort( A, mid+1, r )
    return merge( A1, A2 )
```

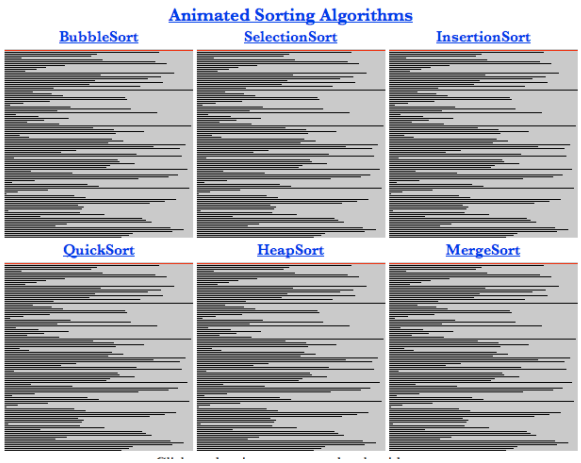


```
def merge(a, b):
    if len(a) == 0:  return b
    if len(b) == 0:  return a

    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] ); i += 1
    while j < len(b):
        result.append( b[j] ); j += 1

    return result
```

Eigenschaften

- Algorithmus ist sehr übersichtlich und einfach zu implementieren
- Aufwand: $n \cdot \log(n)$
  - $\log(n)$ viele Etagen, Aufwand pro Etage in $O(n)$, gilt auch im worst case
  - Nicht besonders schnell, da viel umkopiert wird
- Optimierung:
  - Ständiges Anlegen und Aufgeben 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

S.a.: http://www.sorting-algorithms.com/

What different sorting algorithms sound like

http://www.youtube.com/watch?v=t8g-iYGHpEA

Weitere Optimierungen

- Beobachtung:
  - Arrays auf den unteren Levels der Rekursion sind "klein" und "fast" sortiert
  - Idee: verwende dafür Algo, der auf "fast" sortierten Arrays schneller ist → Insertionsort oder Bubblesort
- Alle Implementierungen von "komplexen" (rekursiven) Sortierverfahren schalten bei $n < b$ auf einen einfachen n^2-Algo um ($b \approx 20$)

Introsort (introspective sorting) [Musser 1997]

- Idee: Algo beobachtet sich selbst, und schaltet auf anderes Verfahren um, wenn er feststellt, dass er einen "Killer-Input" bekommen hat
- Hier konkret:
  - Verwende Quicksort
  - Zähle Rekursionstiefe mit
  - Schalte auf Heapsort um, falls Tiefe > 2 log(n) wird
- Dieser Algo ist z.B. in der C++ STL impl.
- Laufzeiten unter Java: