7. Sortieren

Umordnen von Objekten $a_1, ..., a_n$ in eine Folge $a_{i_1},...,a_{i_n}$, so daß bezgl. einer Ordnung ≤ gilt: $a_{i_j} \le a_{i_{j+1}}, 1 \le j < n$

Begriffe:

1) **Stabilität**: relative Reihenfolge gleichrangiger Objekte wird nicht verändert, d. h.

   $i_j < i_k$, falls $a_{i_j} = a_{i_k}$ und $j < k$.

   Wichtig für vorsortierte Datenmengen! Z. B. erst Sortierung nach Vornamen, dann Sortierung nach Namen → Sortierung nach Vornamen soll erhalten bleiben.

2) **Internes Sortieren**: Datenmenge während des Sortierens vollständig im Hauptspeicher.

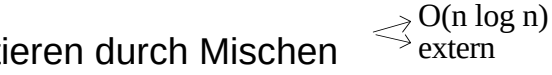
   Externes Sortieren: Datenmenge überwiegend auf externen Speichern.

3) **Zeitbedarf**: Gezeigt: worst-case mindestens O(n log n) Vergleiche.

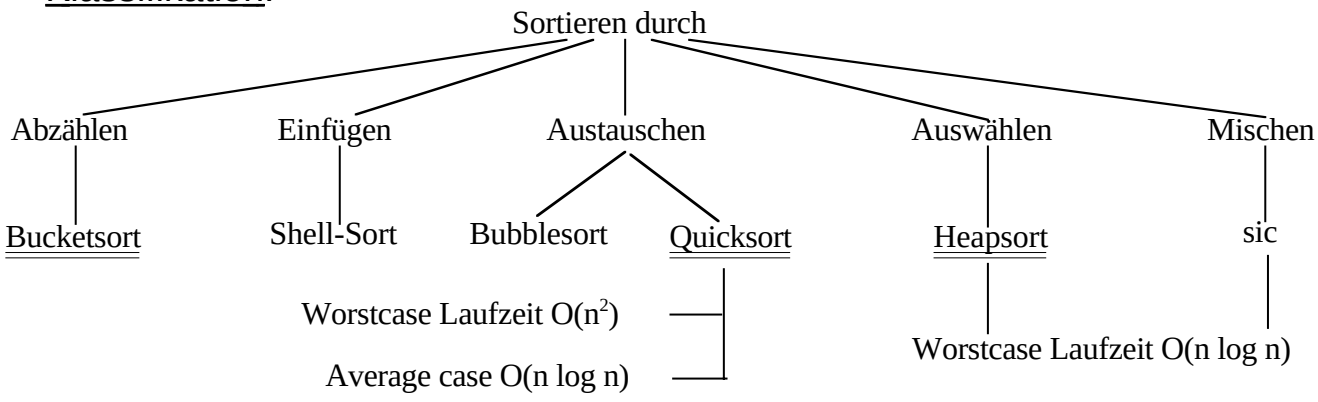
   Forderung: Nur Sortierverfahren betrachten, die diese Schranke erreichen.

   Platzbedarf: Wenig zusätzlicher Speicher (Sortieren **in-situ**)

Beispiele: Bubblesort 

, Sortieren durch Mischen 

Klassifikation:



7.1 Quicksort

Gebräuchlichstes internes Sortierverfahren. (Hoare 1962)

Idee: Gegeben lineares Array A mit n Elementen

a) Transportiere die $\frac{n}{2}$ kleinsten Elemente in die Feldelemente A[1], ..., A$\left[\frac{n}{2}\right]$

und die $\frac{n}{2}$ größten in die Feldelemente $A\left[\frac{n}{2}+1\right],...,A[n]$

b) Wende das Verfahren rekursiv auf die erste Hälfte an.

c) Wende das Verfahren rekursiv auf die zweite Hälfte an.

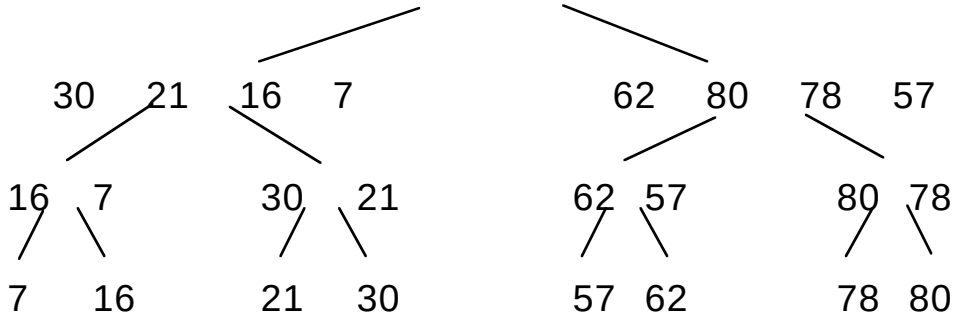
Definition.

Sei M = {m_1, ..., m_n} eine Menge mit einer Ordnung ≤. Das Objekt x ∈ M mit der Eigenschaft

$$|\{m_i \mid m_i < x\}| \approx |\{m_i \mid x < m_i\}|$$

heißt **Median**.

Beispiel: 30 62 21 16 80 7 78 57



Algorithmus:

```
procedure quick (l, r: 1...n);
vor i, j: 1...n; x, w: Objekt;
begin  i:=l; j:=r;
    "Bestimme Median x in A[l] ... A[r]";
    repeat
        while A[i] < x do i:= i+1;
        while A[j] < x do j:= j-1;
        if i ≤ j then
        begin
            "tausche A[i] und A[j]"; i:= i+1; j:= j-1
        end
    until i > j;
    if l < j then quick (l,j);
    if i < r then quick (i,r);
end
```

Laufzeit: Median teilt A jeweils in zwei gleichgroße Hälften.

$O(\log_2 n)$ Teilungsvorgänge --> Arrays nur noch einelementig, Verfahren bricht ab. Der übrige Rumpf von quick benötigt

$O((r-l) + M(r-l)) = O(n+M(n))$ Zeit,

wobei M(n) Zeit für Bestimmen des Medians im Feld der Länge n ist.

Folglich

$$T_{quick}(n) = 2 \cdot T_{quick}\left(\frac{n}{2}\right) + O(n+M(n))$$

Problem: M(n) ?

Median ist zwar in linearer Zeit zu bestimmen (s. Buch von Ottmann/Widmayer), Verfahren jedoch aufwendig.

$$\Rightarrow T_{quick}(n) = O(n) + 2 \cdot T_{quick}\left(\frac{n}{2}\right) = O(n \log n)$$

Abschwächung: Wähle nicht in jedem Rekursionsschritt den Median, sondern irgendein Element x.

Hoffnung: x teilt Feld in zwei etwa gleichgroße Teile. (analoge Idee beim AVL-Baum: „Ein weniger ausgeglichener Baum ist genauso gut wie ein exakt ausgeglichener.“ Hier bezieht sich jetzt die Ausgeglichenheit auf den Rekursionsbaum).

Laufzeit dann:

$T(n) = O(n) + T(m) + T(n-m)$

m und n-m sind hier die Längen der beiden Teile des Arrays.

Sonderfälle:

- Es wird jeweils der Median gewählt:

  $$\Rightarrow T(n) = O(n) + 2 \cdot T\left(\frac{n}{2}\right) = O(n \log n)$$

- Es wird jeweils das kleinste (oder größte) Element für x gewählt:

  $$\Rightarrow T(n) = O(n) + T(1) + T(n-1) = O(n^2)$$

  Quicksort wird „Slowsort“ mit unbrauchbarer Laufzeit (**worst case**). Dieser Fall tritt oft bei schon sortiertem Feld ein.

Frage: Wie oft tritt der worst case auf, wenn x in jedem rekursiven Aufruf zufällig gewählt wird?

Annahmen:

1) Zu sortieren sind die Zahlen 1, ..., n.
2) Alle n! Anordnungen der Zahlen im Ausgangsfeld sind gleich wahrscheinlich
3) Als Median wählen wir immer a_n.

Folgerung: Wird quick für a_1, ..., a_n aufgerufen, so folgt aus den Annahmen, daß jedes k mit $1 \leq k \leq n$ mit gleicher Wahrscheinlichkeit 1/n an Position a_n auftritt und daher als Median gewählt wird.
Anschließend werden zwei Folgen der Längen k-1 und n-k erzeugt.

Zeige nun: Teilfolgen wieder zufällig.

Rekursionsformel:

$T(0) = 0 , T(1) = c$

$$T(n) \leq \underbrace{\frac{1}{n} \sum_{k=1}^{n} (T(k-1)+T(n-k))}_{\text{Mittelwert des Aufwands für alle Teilungen}} + \underbrace{b \cdot n}_{\text{Aufwand für Teilung}}$$

$$n \geq 2: T(n) \leq \frac{2}{n} \sum_{k=1} T(k) + b \cdot n$$

Zeige nun: $T(n) \leq c \cdot n \log n$ für hinreichend großes c.

Beweis durch Induktion:

Induktionsanfang: n=1,2 ist klar.

Induktionsvoraussetzung: Sei die Aussage wahr für alle i<n, also T(i) = c · i · log i für i<n

Dann gilt:

$$T(n) \le \frac{2}{n}\sum_{k=1}^{n-1} T(k) + b \cdot n = \frac{2c}{n}\sum k \cdot \log k + b \cdot n$$

$$= \frac{2c}{n}\sum_{k=1}^{n/2}\underbrace{[k \cdot \log k]}_{\le \log (n-1)} + \sum_{k=1}^{n/2-1}\left[\left(\frac{n}{2}+k\right)\underbrace{\log\left(\frac{n}{2}+k\right)}_{\le \log (n-1)}\right] + b \cdot n$$

$$\le \frac{2c}{n}\left[\frac{n}{4}\left(\frac{n}{2}+1\right)\log n - \frac{n^2}{8} - \frac{n}{4} + \left(\frac{3n^2}{8} - \frac{3n}{4}\right)\log n\right] + bn$$

$$= \frac{2c}{n}\left[\left(\frac{n^2}{2} - \frac{n}{2}\right)\log n - \frac{n^2}{8} - \frac{n}{4}\right] + bn$$

$$= c \cdot n \log n - \underbrace{c \cdot \log n}_{\ge 0} - \frac{cn}{4} - \frac{c}{2} + bn$$

$$\le c \cdot n \log n - \frac{cn}{4} - \frac{c}{2} + bn$$

Für c ≥ 4b: T(n)≤ c n · log n.

Ergebnis: Im Mittel benötigt Quicksort O(n log n) Vergleiche zum Sortieren von n Zahlen.

Praxis: Wähle Median

- aufgrund einer Stichprobe
- zufällig
- als Median von (a_l, $a_{(l+r)/2}$, a_r)

Speicher: wird vor allem durch die Rekursionstiefe bestimmt:

- im schlimmsten Fall: O(n)
- im mittleren Fall: O(log n)

Durch geschickte Implementierung läßt sich der Stack auch im worst-case auf O(log n) begrenzen.

7.2 Bucketsort

Sortierverfahren, das ohne Vergleiche auskommt und besondere Voraussetzungen an die zu sortierenden Objekte stellt.

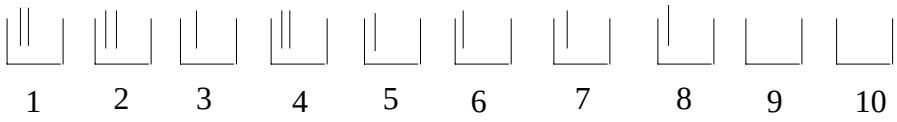
Idee:

- Lege für jedes zu sortierende Objekt ein Bucket (Behälter, Eimer, Fach) an
- Lege jedes Objekt in das zugehörige Bucket (gleiche Objekte kommen in denselben Bucket)
- Gib die Buckets der Ordnung nach aus.

Voraussetzung: Grundtyp der zu sortierenden Elemente endlich (nicht zu groß).

Beispiel: Grundtyp 1 ... 10

Eingabe: 7 2 4 2 8 1 1 3 6 4 5



Ausgabe: 1 1 2 2 3 4 4 5 6 7 8

Algorithmus: Skalarer Grundtyp T=(t_1, ..., t_m)

```
var buckets: array [T] of integer;
for i:= t1 to tm do buckets [ti] : = 0;
while not eof do
begin
    read (x); buckets [x] : = buckets [x] +1
end;
for i : =t1 to tm do
    while buckets [ti] > 0 do
    begin
        write (ti);
        buckets [ti] : = buckets [ti] -1
    end
```

Laufzeit: O(m+n+m) = O(m+n)

m konstant --> Laufzeit O(n)

Speicher: O(m) (bzw. O(1)).

Anwendung:

sortieren von Zeichenfolgen (**Radixsort**), in lexikographischer Reihenfolge

z.B. acd, bad, dca, dba, bcd

Vorgehensweise:

- Anlegen von 26 Buckets für die Buchstaben (+1 Bucket für Leerzeichen)
- Bucketsort beginnend mit dem letzten Zeichen so oft aufrufen, wie das längste Wort lang ist.

1)

dca, dba, acd, bad, bcd

2)

bad, dba, dca, acd, bcd

3)

acd, bad, bcd, dba, dca STOP

Bemerkung: Früher wurde gezeigt, daß zum Sortieren mind. O(n log n) Vergleiche erforderlich sind. Bucketsort kommt ganz ohne Vergleiche aus. Wie paßt das zusammen?

Klar, Bucketsort ist **kein allgemeines** Sortierverfahren. Es nutzt zusätzliche Eigenschaften des zu sortierenden Raumes aus; nicht jede zu sortierende Menge besitzt diese Eigenschaften. kein Widerspruch zwischen beiden Aussagen.

Wichtig: Komplexitätsaussagen sind immer relativ zu den zur Verfügung stehenden elementaren Operationen zu interpretieren.

7.3 Heapsort

Merkmale:

- Sortieren durch Auswählen
- garantiertes O(n log n)-Verfahren
- im Mittel schlechter als Quicksort (es gibt allerdings auch bessere Implementierungen)

Idee:

Wähle in jedem Auswahlschritt das Maximum des verbleibenden Feldes und gib es aus.
Geschickte Auswahl --> O(n log n)-Verfahren
Zentrale Datenstruktur: binärer Baum als sog. **Heap** organisiert.

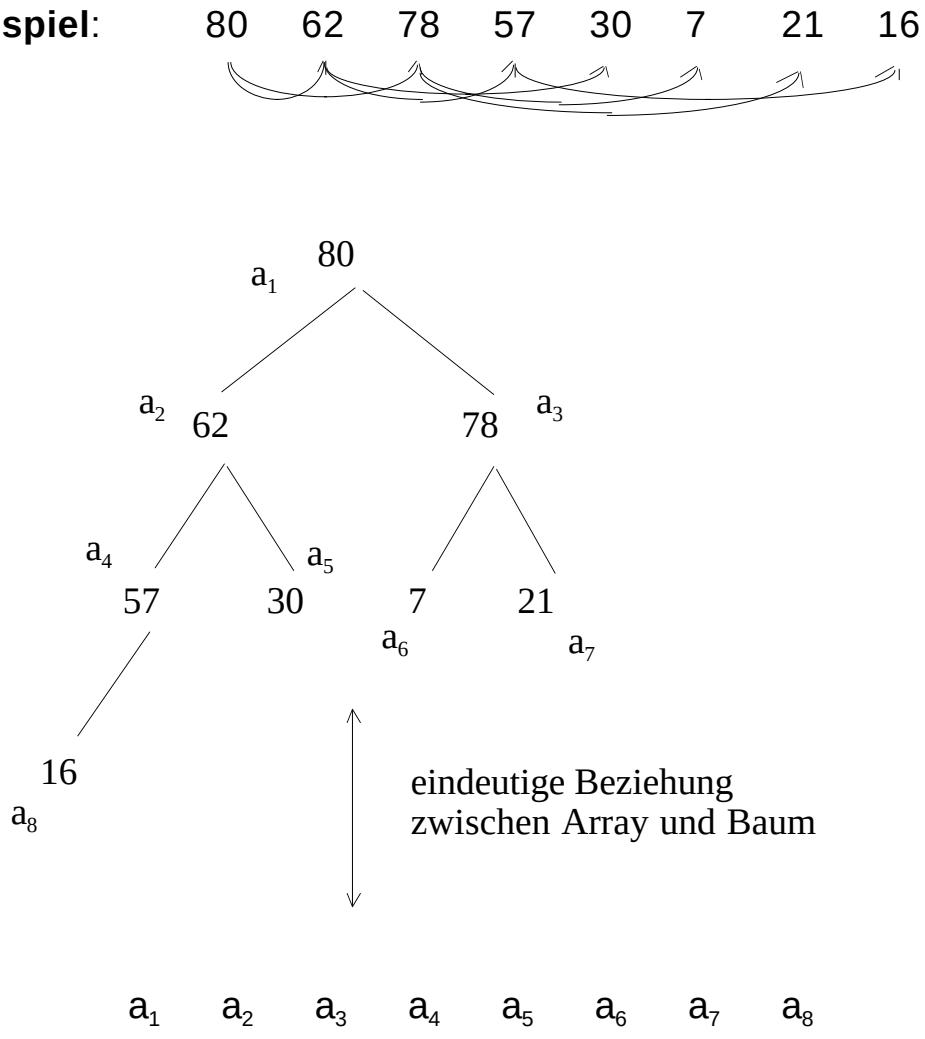
Definition

Eine Folge von Objekten $a_1, ..., a_n$ heiß **Heap**, falls gilt:

$a_j \geq a_{2j}$, $a_j \geq a_{2j+1}$, für $2j, 2j+1 \leq n$

Folgerung: $a_1 = \max_{1 \leq j \leq n} a_j$

Beispiel:



Im folgenden häufig Wechsel zwischen Argumentation auf Bäumen und Arrays.

Grundgerüst: Algorithmus Heapsort:

Annahme: Heap aus a_1, ..., a_n gegeben.

Methode: <u>while</u> Heap nicht leer <u>do</u>

Gib a_1 aus;

Entferne a_1 aus dem Heap;

Stelle Heap-Eigenschaft für die Schlüssel a_2, ..., a_n im Indexbereich 1 ...n-1 wieder her

danach steht also das max (a_2, ..., a_n) an Position 1. ←

<u>od</u>.

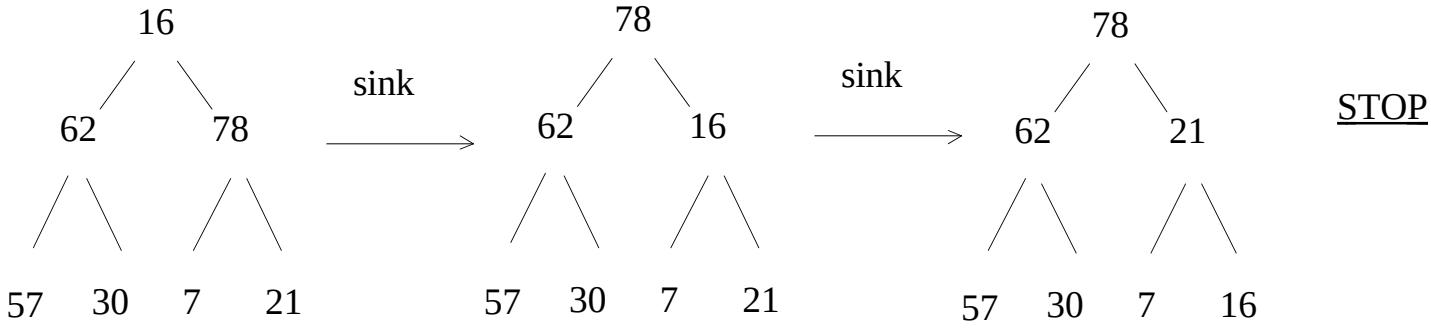
Problem: Wiederherstellen der Heap-Bedingung:

Nach Entfernen von a_1 --> zwei Heaps mit Wurzel a_2 und a_3.

Vorgehen: Setze Element mit höchstem Index an Positon 1 --> meist Heap-Eigenschaft verletzt.

Lasse Element im Heap **absinken**, indem es solange immer wieder mit dem größeren der beiden Söhne vertauscht wird, bis beide Söhne kleiner sind oder das Element unten angekommen ist.

Beispiel: Aus dem Heap oben wird 80 entfernt und 16 an seine Stelle gesetzt.



Algorithmus: Wiederherstellen der Heap-Eigenschaft: für a_1, ... a_{m+1}

```
Setze a_{m+1} an die Stelle von a_1;
while a_i hat linken Sohn do
    if a_i hat rechten Sohn then
        a_j :=Sohn von a_i mit größerem Wert
    fi;
    if a_i < a_j then
        vertausche a_i und a_j ; i:=j
    else
        STOP
    fi
od.
```

Für endgültige Implementierung ergänze: Aufbau des ersten Heaps.
Klar: Versenke die zu sortierenden Elemente nacheinander im Heap.

Das Programm: [Floyd 1964]

```
procedure sink (i, t: integer);   <-- Absinken des Elements i im Bereich 1..t
var j, k: integer;
begin
    j:=2·i; k:=j+1;
    if j≤ t then
    begin
        if A[i] ≥ A[j] then j:= i;
            if k ≤ t then
                if A[k] > A[j] then j:=k;
            if i≠ j then
            begin
                „vertausche A[i] und A[j]“;
                sink (j, t)
            end
    end
end.
```

Hauptprogramm Sortieren:

```
    {Aufbau eines Heap}
    for i:=(n div 2) downto 1 do sink (i, n);
    {Sortieren}
    for m:=n downto 2 do
    begin
        „vertausche A[1] und A[m]“;
        sink (1, m-1)
    end
```

- Nutze, daß die Elemente

  $a_{\lfloor n/2 \rfloor+1}, \ldots, a_n$

  bereits die Heap-Eigenschaft besitzen.
  Daher beginnt die Schleife bei i=n div 2.

Analyse:

- Laufzeit von sink bestimmt Laufzeit von Heapsort.
- Versickern bis max. zu den Blättern durch fortlfd. Vertauschen.
- Bei n Objekten beträgt die Höhe des Baumes O(log n).
- Jeder Versickerungsprozeß benötigt O(log n) Schritte im worst-case.
- Aufbau eines Heaps also O(n log n) [tatsächlich: O(n), wenn man genauer nachrechnet]
- Das Sortieren ebenfalls O(n· log n), da genau n Versickerungsschritte auszuführen sind.

Bemerkungen:

- Heapsort ist echtes in-situ-Verfahren, da nur konstant viel Speicher zusätzlich benötigt wird (, anders als die gängigen Varianten von Quicksort).
- Vorsortierung (anders als bei gängigen Versionen von Quicksort) der Daten spielt keine Rolle.
- Heapsort ist **nicht** stabil.