

6. Sortieren

Umordnen von Objekten a_1, \dots, a_n in eine Folge a_{i_1}, \dots, a_{i_n} , so daß bezgl. einer Ordnung \leq gilt:

$$a_{i_j} \leq a_{i_{j+1}}, 1 \leq j < n$$

Begriffe:

- 1) Stabilität: Ein Sortierverfahren heißt stabil, falls die relative Reihenfolge gleichrangiger Objekte nicht verändert wird, d. h.

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

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

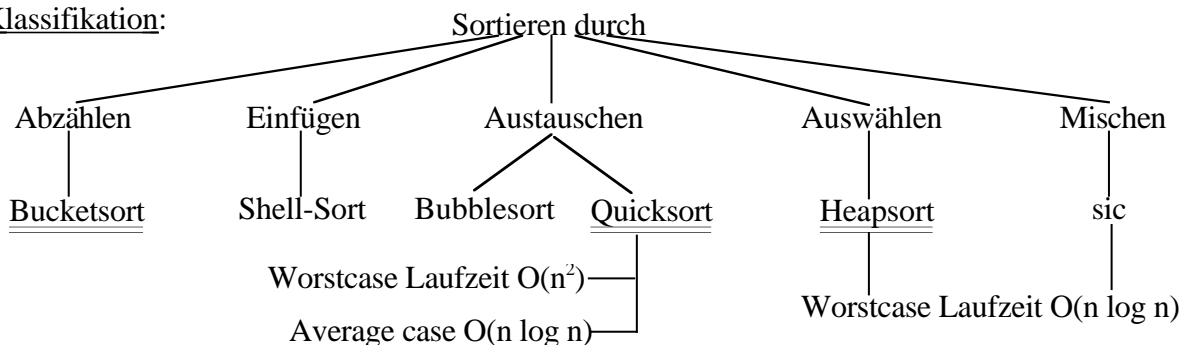
- 2) Internes Sortieren: Datenmenge befindet sich während des Sortierens vollständig im Hauptspeicher.

Externes Sortieren: Der größte Teil der Datenmenge befindet sich auf externen Speichern.

- 3) Zeitbedarf: Gem. Kap. 4 werden im worst-case mindestens $O(n \log n)$ Vergleiche benötigt.
 Forderung: Nur Sortierverfahren betrachten, die diese Schranke erreichen. Platzbedarf: Wenig zusätzlicher Speicher (Sortieren in-situ)

Beispiele: Bubblesort $\begin{cases} \text{intern} \\ \text{ineffizient} \end{cases}$, Sortieren durch Mischen $\begin{cases} O(n \log n) \\ \text{extern} \end{cases}$

Klassifikation:



6.1 Quicksort

Gebräuchlichstes internes Sortierverfahren. (Hoare 1962)

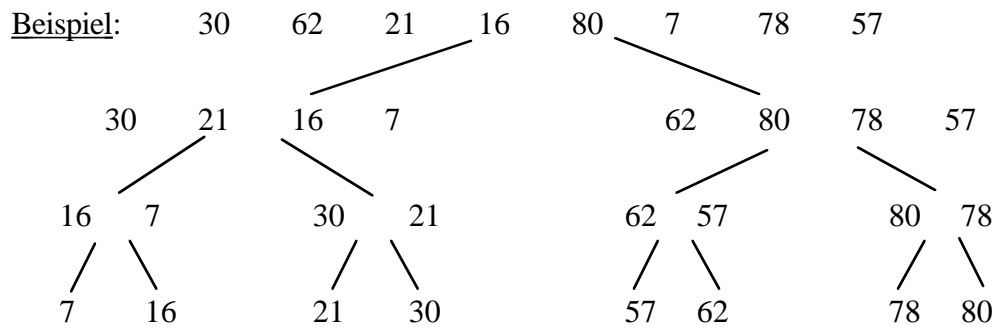
Idee: Gegeben lineares Array A mit n Elementen

- a) Transportiere die $\frac{n}{2}$ kleinsten Elemente in die Feldelemente $A[1], \dots, A[\frac{n}{2}]$ und die $\frac{n}{2}$

größten in die Feldelemente $A[\frac{n}{2} + 1], \dots, A[n]$

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

Def. Sei $M = \{m_1, \dots, m_n\}$ eine Menge mit einer Ordnung \leq . Das Objekt $x \in M$ mit der Eigenschaft $|\{m_i \mid m_i < x\}| \approx |\{m_i \mid x < m_i\}|$ heißt Median.



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
  end

```

Laufzeit: Durch den Median wird A jeweils in zwei gleichgroße Hälften geteilt. Nach $O(\log_2 n)$ Teilungsvorgängen sind die entstehenden Arrays nur noch einelementig, und das Verfahren bricht ab. Der übrige Rumpf von quick benötigt

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

wobei $M(n)$ die Zeit für das Bestimmen des Medians in einem Feld der Länge n ist. Folglich

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

Problem: $M(n)$? Der Median ist zwar in linearer Zeit zu bestimmen (s. Buch von Ottmann/Widmayer), das Verfahren ist jedoch aufwendig.

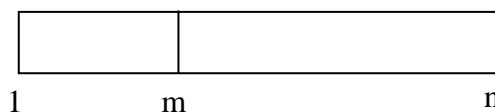
$$\Rightarrow T_{\text{quick}}(n) = O(n) + 2 \cdot T_{\text{quick}}\left(\frac{n}{2}\right) = O(n \log n)$$

Abschwächung: Wähle nicht in jedem Rekursionsschritt den Median, sondern irgendein Element x. Wird x zufällig gewählt, so ist anzunehmen, daß es das Feld in zwei etwa gleichgroße Teile teilt. (Eine analoge Idee haben wir schon beim AVL-Baum benutzt: „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:

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

In diesem Falle wird also Quicksort zum „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, \dots, 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.

Ist k Median, so werden durch die Aufteilung zwei Folgen der Längen k-1 und n-k erzeugt.

Man kann nun zeigen, daß die Teilfolgen wieder zufällig sind.

Rekursionsformel:

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

$$T(n) \leq \frac{1}{n} \sum_{k=1}^n (T(k-1) + T(n-k)) + b \cdot n$$

Mittelwert des Aufwands für alle Teilungen Aufwand für Teilung

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

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 $T(i) = c \cdot i \cdot \log i$ für $i < n$

Dann gilt:

$$\begin{aligned}
 T(n) &\leq \frac{2}{n} \sum_{k=1}^{n-1} T(k) + b \cdot n = \frac{2c}{n} \sum_{k=1}^{n-1} k \cdot \log k + b \cdot n \\
 &= \frac{2c}{n} \sum_{k=1}^{n/2} \underbrace{[k \cdot \log k]}_{\leq \log_{n-1}} + \sum_{k=1}^{n/2-1} \left[\left(\frac{n}{2} + k \right) \log \left(\frac{n}{2} + k \right) \right]_{\leq \log_{n-1}} + b \cdot n \\
 &\leq \frac{2c}{n} \left[\frac{n}{4} \left(\frac{n}{2} + 1 \right) \log n - \frac{n^2}{8} - \frac{n}{4} + \left(\frac{3n^2}{8} - \frac{3n}{4} \right) \log n \right] + bn \\
 &= \frac{2c}{n} \left[\left(\frac{n^2}{2} - \frac{n}{2} \right) \log n - \frac{n^2}{8} - \frac{n}{4} \right] + bn \\
 &= c \cdot n \log n - \underbrace{c \cdot \log n}_{\geq 0} - \frac{cn}{4} - \frac{c}{2} + bn \\
 &\leq c \cdot n \log n - \frac{cn}{4} - \frac{c}{2} + bn
 \end{aligned}$$

Für $c \geq 4b$: $T(n) \leq c n \cdot \log n$.

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 Stach auch im worst-case auf $O(\log n)$ begrenzen.

6.2 Bucketsort

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

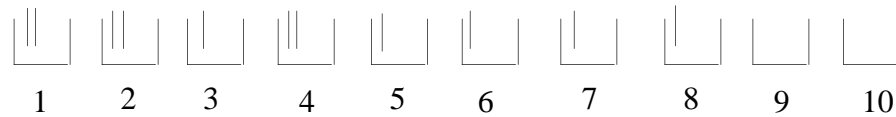
Idee: • Lege für jedes zu sortierende Objekt ein Bucket (Behälter, Eimer, Fach) an

- Lege jedes Objekt in das zugehörige Bucket (gleiche Objekte kommen in denselben Bucket)
- Gib die Buckets der Ordnung nach aus.

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

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 $T=(t_1, \dots, t_m)$

```

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

Da m konstant, folgt Laufzeit $O(n)$

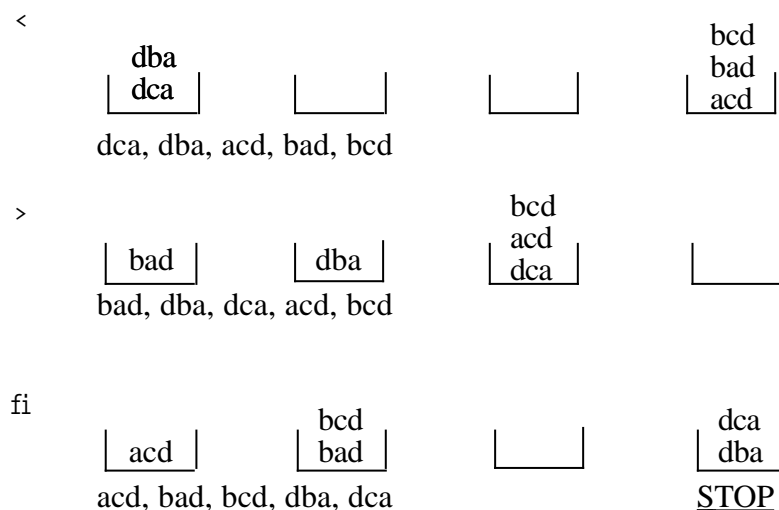
Speicher: $O(m)$ (bzw. $O(1)$).

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.



Bemerkung: In Kap. 4 wurde gezeigt, daß zum Sortieren mind. $O(n \log n)$ Vergleiche erforderlich sind. Bucketsort kommt ganz ohne Vergleiche aus.

Wie geht 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. \bar{O} kein Widerspruch zwischen beiden Aussagen.

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

6.3 Heapsort

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

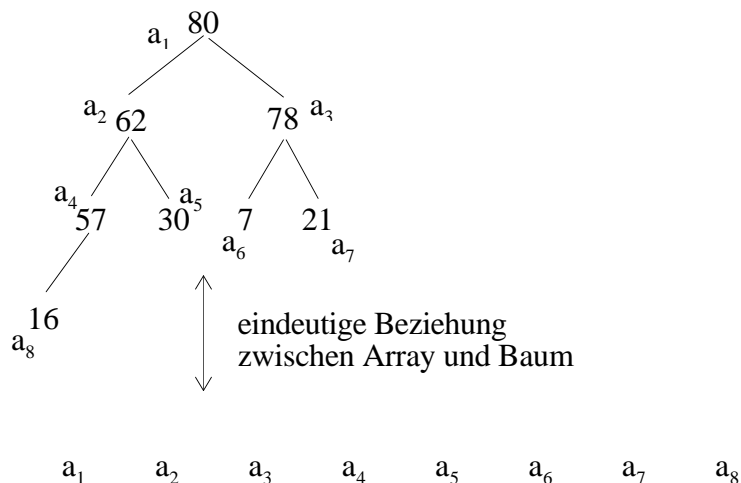
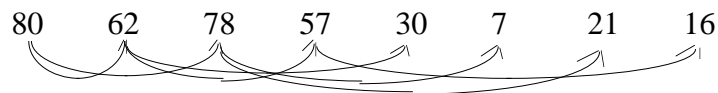
Idee: Man wählt in jedem Auswahlsschritt das Maximum des verbleibenden Feldes und gibt es aus. Die Auswahl ist so geschickt zu organisieren, daß ein $O(n \log n)$ -Verfahren entsteht. Man bedient sich dazu - wie oft - eines binären Baums, den man als sog. Heap organisiert.

Def.: Eine Folge von Objekten a_1, \dots, a_n heißt Heap, falls gilt:

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

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, \dots, a_n gegeben.

Methode: Solange Heap nicht leer tue

Gib a_1 aus;

Entferne a_1 aus dem Heap;

danach steht also \leftarrow Stelle Heap-Eigenschaft für die Schlüssel a_2, \dots, a_n im Indexbereich $1 \dots n-1$
das $\max(a_2, \dots, a_n)$ wieder her

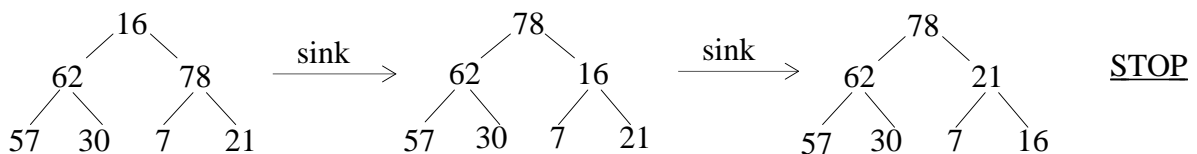
an Position 1. ende.

Problem Wiederherstellen der Heap-Bedingung:

Offenbar liegen nach Entfernen von a_1 zwei Heaps mit Wurzel a_2 und a_3 vor.

Vorgehen: Man setzt das Element mit dem höchsten Index an die Position 1. Dies verletzt im allgemeinen die Heap-Eigenschaft. Dann läßt man das Element im Heap absinken, indem man es solange immer wieder mit dem größeren der beiden Söhne vertauscht, 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, \dots, a_{m+1}

Setze a_{m+1} an die Stelle von a_1 ;

Solange a_1 hat linken Sohn tue

wenn a_1 hat rechten Sohn dann

$a_j :=$ Sohn von a_1 mit größerem Wert;

wenn $a_1 < a_j$ dann

vertausche a_1 und a_j ; $i := j$

sonst

STOP

ende

ende.

Für die endgültige Implementierung fehlt noch der Aufbau des ersten Heaps. Die Vorgehensweise ist klar: Man versenkt die zu sortierenden Elemente nacheinander im Heap.

Das Programm: [Floyd 1964]

procedure sink (i, t : integer); \leftarrow Absinken des Elements i im Bereich $1..t$

var j, k : integer;

begin

$j := 2 \cdot i$; $k := j + 1$;

if $j \leq t$ then

begin

if $A[i] \geq A[j]$ then $j := i$;

if $k \leq t$ then

if $A[k] > A[j]$ then $j := k$;

if $i \neq j$ 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
```

' Hier wird ausgenutzt, daß die Elemente

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

bereits jeweils die Heap-Eigenschaft besitzen. Daher beginnt die Schleife bei $i = n \text{ div } 2$.

Analyse:

Die Laufzeit von Heapsort wird wesentlich durch die Laufzeit von sink bestimmt. Beim Versickern werden Elemente bis max. zu den Blättern durch fortld. Vertauschen transportiert. Bei n Objekten beträgt die Höhe des Baumes $O(\log n)$. Jeder Versickerungsprozeß benötigt also $O(\log n)$ Schritte im worst-case.

Der Aufbau eines Heaps benötigt also

$O(n \log n)$ [tatsächlich: $O(n)$, wenn man genauer nachrechnet]

Das Sortieren ebenfalls $O(n \cdot \log n)$, da genau n Versickerungsschritte auszuführen sind.

Bemerkungen:

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