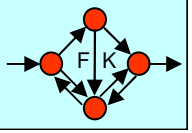
Sortiermethoden	Zeitaufwand	zusätzl. Platz
Aussuchen / Auswählen		
a. Minimumsuche	$\frac{1}{2} \cdot n^2$	konstant
b. Heapsort	$\leq 2n \cdot \log(n)$	konstant
Einfügen		
a. Einfügen in Listen	$\frac{1}{2} \cdot n^2$	konstant
b. Baumsortieren (AVL-Bäume)	$\leq 1{,}4404 \cdot n \cdot \log(n)$	O(n)
c. Fachverteilen (im Mittel)	$O(n \cdot \log(n))$	O(n)
Austauschen		
a. Benachb. Austauschen	$\frac{1}{2} \cdot n^2$	konstant
b. Shellsort	$O(n^{\frac{3}{2}})$	$\leq \log(n)$
c. Quicksort (im Mittel)	$1{,}3863 \cdot n \cdot \log(n)$	$2 \log(n)$
Mischen		
Verschmelzen (merge sort)	$O(n \cdot \log(n))$	n
Streuen und Sammeln	O(n)	O(n)



3.4.1.12 Überblick über die üblichen Sortiermethoden:

Aussuchen / Auswählen:

a. Minimumsuche (minimum sort)

b. Heapsort (normal, bottom up, ultimativ)

Einfügen:

a. Einfügen in Listen (Insertion sort)

b. Baumsortieren (mit binären Bäumen, AVL-Bäumen, ...)

c. Fachverteilen (und radix exchance)

Austauschen:

a. Benachbartes Austauschen (bubble sort, shaker sort)

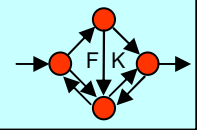
b. Shellsort

c. Quicksort

Mischen:

merge sort und diverse Varianten

Streuen und Sammeln (bucket sort)



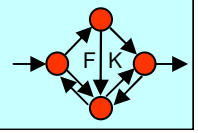
3.4.2 Sortieren durch Aussuchen/Auswählen

Vorgehen:
Wähle das kleinste Element aus, stelle es an die erste Stelle
und mache genauso mit den restlichen Elementen weiter.

3.4.2.1 Sortieren durch "Minimum sortieren":

```
for i in 1..n-1 loop
    min := A(i); pos := i;          -- finde das kleinste Element von A(i) bis A(n)
    for j in i+1..n loop
        if A(j) < min then min:=A(j); pos := j; end if;
    end loop;                       -- das kleinste Element steht an Position pos
    A(pos) := A(i); A(i) := min;    -- nun steht das kleinste Element an Position i
end loop;
```

Zahl der Vergleiche stets ½·n·(n-1) Schritte: $\Theta(n^2)$
Platzaufwand 4 zusätzliche Speicherplätze: $O(1)$



Aufgabe 1 (Selectionsort)

Sortieren Sie die Folge

7	4	2	8	9	7	7	2	1

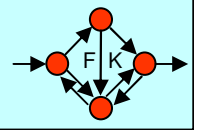
mit Hilfe von Selectionsort. Welche Komplexität hat das Sortierverfahren im besten / schlechtesten Fall?

Aufgabe 2 (Bubblesort)

Sortieren Sie die Folge

2	5	1	3	6	4

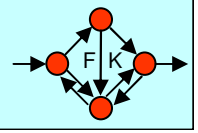
mit Hilfe von Bubblesort und machen Sie sich jeden Schritt klar. Berechnen Sie den Aufwand von Bubblesort für den besten / schlechtesten Fall.



3.4.4.1 Bubble Sort

3.4.4.1 Bubble Sort für ein Integer-Feld A mit dem Indextyp 1..n (das Feld A sei vom Feld-Typ Vektor):

```
procedure BubbleSort (A: in out Vektor) is
Weiter: Boolean := True;  H: Integer;
begin
    while Weiter loop
        Weiter := False;
        for i in 1..n-1 loop
            if A(i) > A(i+1) then Weiter := True;
                H := A(i); A(i) := A(i+1); A(i+1) := H;
            end if;
        end loop;
    end loop;
end BubbleSort;
```



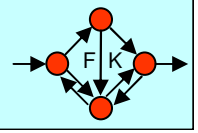
Worst case / Quicksort

Satz 3.4.1.11: Ein Sortierverfahren, das ausschließlich auf Vergleichen zweier Elemente beruht, benötigt im worst case

$$\text{mindestens } \log(n!) \approx n \cdot \log(n) - 1{,}4404 \cdot n$$

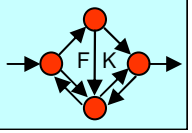
Schritte.

1.7.3.3: *Quicksort.* Man wählt ein "Pivot"-Element p aus und spaltet die in einem array gespeicherte Folge in zwei Teilfolgen, von denen alle Elemente der ersten Teilfolge kleiner oder gleich p und alle Elemente der zweiten Teilfolge größer oder gleich p sind. Danach: rekursiv weiter mit beiden Teilfolgen, sofern die jeweilige Teilfolge noch mindestens zwei Elemente besitzt.



3.4.4.2 Quicksort: Aus 1.7.3.4 übernehmen wir mit den Datentypen *type Index is 1..n; ... A: array (Index) of Integer;* ... das Programm:

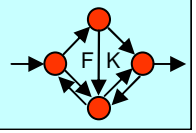
```
procedure Quicksort (L, R: Index) is     -- A ist global, A(L..R) wird sortiert
i, j: Index; p, h: Integer;              -- p wird das Pivot-Element
begin
  if L < R then
     i := L; j := R; p := A((L+R)/2);    -- man kann p auch anders wählen
     while i <= j loop                   -- die Indizes i und j laufen aufeinander zu
          while A(i) < p loop i := i+1; end loop;
          while A(j) > p loop j := j-1; end loop;
          if i <= j then h:=A(i); A(i):=A(j); A(j):=h;
                         i := i+1; j := j-1;  end if;
     end loop;                           -- auch bei Gleichheit A(i)=p oder A(j)=p vertauschen!
     if (j-L) < (R-i) then Quicksort(L, j); Quicksort(i, R);
     else Quicksort(i, R); Quicksort(L, j); end if;  -- Vorsicht; siehe unten!
  end if;
end Quicksort;

... Quicksort(1,n); ...                  -- Aufruf des Sortierverfahrens
```



3.4.4.4 Zeitbedarf von Quicksort:

Zeitbedarf: worst case: $\approx \frac{1}{2} \cdot n^2$ Vergleiche, wenn das Pivot-Element stets das kleinste oder das größte Element des Teilfelds ist.
best case: $n \cdot \log(n)$, wenn das Pivot-Element stets das mittelste der Elemente des Teilfelds ist.
average case: Es ist mit $1.3863 \cdot n \cdot \log(n) - 1{,}8456 \cdot n$ Vergleichen im Mittel zu rechnen. Begründung:

Man kann das Aufteilen des Feldes in zwei Teilfelder mit dem Aufbau eines binären Suchbaums vergleichen, wobei das Pivot-Element in die Wurzel kommt und aus den beiden Teilfeldern rekursiv der linke und rechte Unterbaum aufgebaut werden. Quicksort verhält sich daher im Mittel genau wie das Baumsortieren mit binären Suchbäumen (Satz 3.2.2.15). Die formalen Berechnungen liefern das gleiche Ergebnis, siehe Lehrbücher.



Aufgabe 3 (Quicksort)

Zeigen Sie den Ablauf von Quicksort beim Sortieren der Folge:

13	2	8	12	6	3	9	10	11	4	5	1	7

Wählen Sie als Pivot-Element immer das erste Element des zu sortierenden Teilfeldes.

Welche Komplexität hat das Sortierverfahren im besten / schlechtesten Fall? Welchen Einfluß hat die Wahl des Pivot-Elements und die Vorsortiertheit der Folge auf das Verhalten von Quicksort?

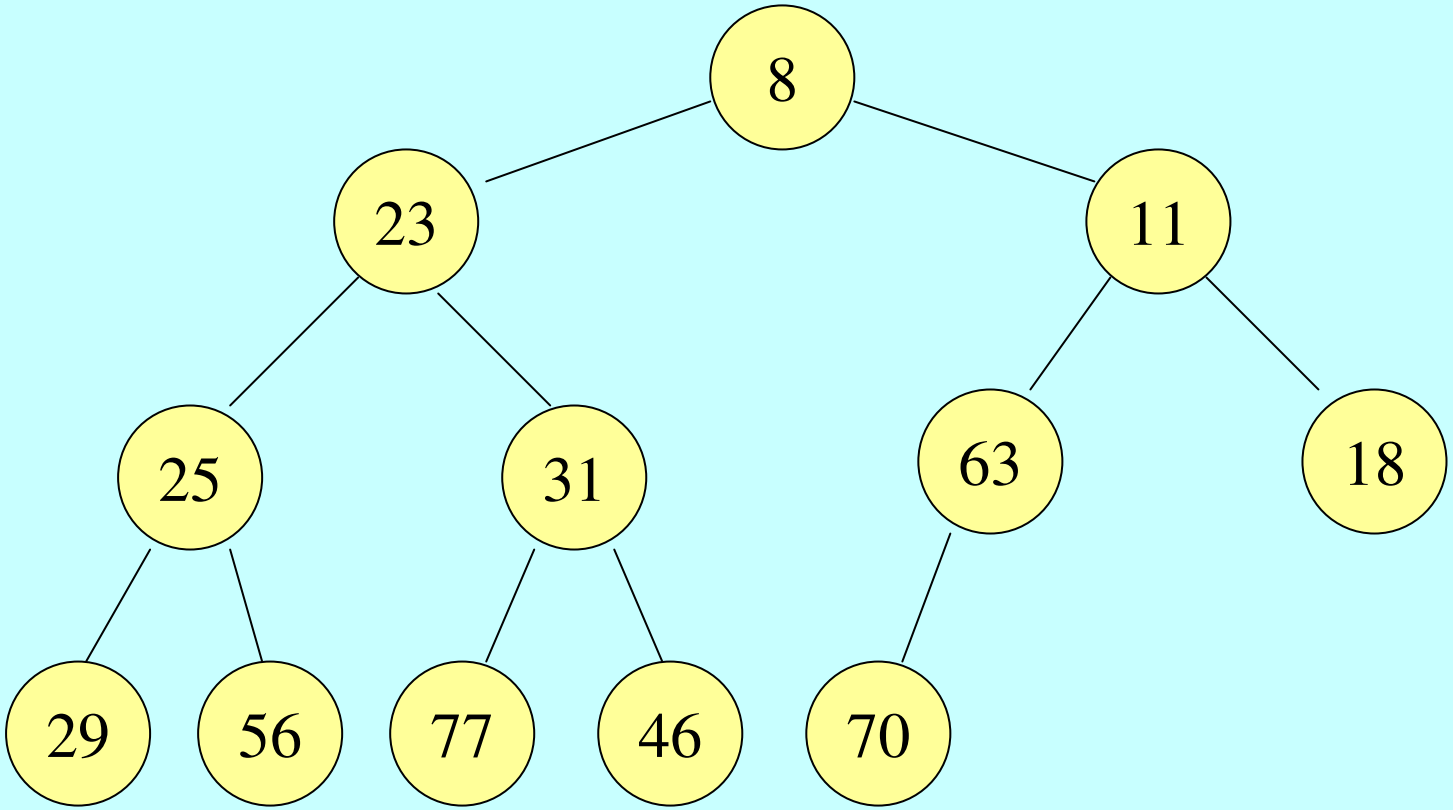
Aufgabe 4 (Heapsort)

Sortieren Sie die Folge

17	23	16	9	45	23	4

mit Hilfe von Heapsort. Erläutern Sie, wie man von der Darstellung als Baumstruktur unter (b) zu der in der Vorlesung verwendeten Heap-Darstellung kommt. Welche Komplexität hat das Sortierverfahren im besten / schlechtesten Fall?

Heap mit zwölf Zahlen:



Als Feld levelweise aufgeschrieben: $A(i) \leq A(2i)$ und $A(i) \leq A(2i+1)$.

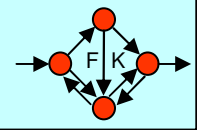
8	23	11	25	31	63	18	29	56	77	46	70
1	2	3	4	5	6	7	8	9	10	11	12

3.4.2.3 Heapsort:

Gegeben sei ein Feld A(1), A(2), A(3), ..., A(n) mit Elementen aus einer geordneten Menge.

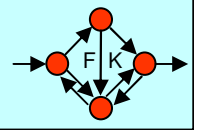
1. Wandle dieses Feld in einen absteigenden Heap um, so dass anschließend gilt: $A(i) \geq A(2i)$ und $A(i) \geq A(2i+1)$ für alle i (sofern $2i \leq n$ bzw. $2i+1 \leq n$ ist).
2. Für j von n abwärts bis 2 wiederhole:
   Vertausche A(1) und A(j).
   (Nun verletzt A(1) in der Regel die Heapeigenschaft.)
   Wandle das Feld A(1..j-1) ausgehend von der Wurzel so um, dass wieder ein absteigender Heap entsteht.

Im Folgenden beschreiben wir das Umwandeln in einen Heap (1.) und die Wiederherstellung der Heap-Eigenschaft (2.).



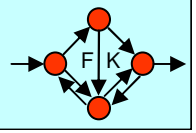
3.4.5.2: Sortieren durch Mischen

Es soll ein Feld A(1..n) durch Mischen sortiert werden.
Zuerst die "*Bottom-Up*-Denkweise", die zu einem Iterations-verfahren führt: Man verschmilzt zunächst je zwei Folgen der Länge 1 zur sortierten Folgen der Länge 2 (man muss hierfür n/2 mal "Verschmelzen" aufrufen),
dann verschmilzt man je zwei sortierte Folgen der Länge 2 zu sortierten Folgen der Länge 4 (n/4 mal "Verschmelzen" aufrufen),
danach das Gleiche für sortierte Folgen der Länge 4, 8, 16 usw., bis zwei Folgen der Läng n/2 zu einer sortierten Folge verschmolzen wurden.
Sofern n eine Zweierpotenz war, funktioniert dieses Verfahren bereits; im allgemeinen Fall muss man beim Verschmelzen jeweils Folgen, deren Länge sich bis zum Faktor 2 unter-scheiden darf, berücksichtigen (vgl. Beispiel).
Dieses Mischen heißt in der Literatur "straight mergesort".



3.4.5.3: Programm zum Sortieren durch Mischen ("Mergesort")

```
procedure Mergesort (L, R: in Integer) is
Mitte: Integer; i, j, k: Integer;          -- Die Felder A und B sind global.
begin                                      -- Es wird A(L..R) sortiert.
if R > L then  Mitte := (L+R)/2;           -- Sortiere rekursiv zwei Hälften der Folge
    Mergesort(L, Mitte);  Mergesort(Mitte+1, R);
    i := L; j := Mitte+1; k := L-1;        -- Nun mischen von A nach B, wie in 3.4.5.1
    while i <= Mitte and j <= R loop  k := k+1;
        if A(i) < A(j)  then B(k) := A(i); i := i+1;
                        else  B(k) := A(j); j := j+1; end if;
    end loop;
    if i <= Mitte  then
        for m in i..Mitte loop k:=k+1; B(k):=A(m); end loop;
    else for m in j..R loop k:=k+1; B(k):=A(m); end loop; end if;
    for m in L..R loop A(m) := B(m); end loop;     -- zurückkopieren nach A
end if;
end Mergesort;
```



Aufgabe 5 (Mergesort)

Sortieren Sie die Folge

7	4	2	8	9	7	7	2	1

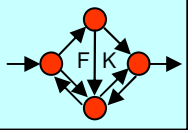
mit Hilfe von Mergesort. Welche Komplexität hat das Sortierverfahren im besten / schlechtesten Fall?

Aufgabe 6 (Bucketsort)

Zeigen Sie den Ablauf von Bucketsort beim Sortieren der Folge:

13	2	13	5	2	13	13	2	4	13	6	2	6

Welche Komplexität hat das Sortierverfahren im besten / schlechtesten Fall?



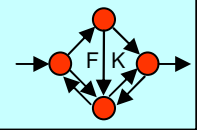
Bucketsort

```
declare A: array (1..n) of Integer;                    -- n ist global
bucket: array (0..m-1) of "liste von Integer"; ...
begin
for j in 0..m-1 loop bucket(j) := "leer"; end loop;
for i in 1..n loop                                     -- streuen
      "hänge A(i) an bucket(A(i)) an";
end loop;
i := 0;
for j in 0..m-1 loop                                   -- sammeln
      while "bucket(j) nicht leer" loop
            i := i+1; A(i) := "erstes Element von bucket(j)";
            "Entferne aus bucket(j) das erste Element"; end loop;
end loop;
end;
```

3.4.3 Sortieren durch Einfügen

Wenn die Elemente einer Folge durch Zeiger (sortierte Listen, Suchbäume) dargestellt werden, so wird man die einzelnen Elemente der Folge nacheinander in diese Struktur einfügen und dabei die Struktur stets wiederherstellen.

Einfachster Fall: Einfügen in eine sortierte Liste.
Eine Liste heißt sortiert, wenn für alle Elemente der Liste gilt: p.Inhalt ≤ p.next.Inhalt. Hiebei ist p ein Zeiger, der auf das jeweilige Element der Liste verweist.

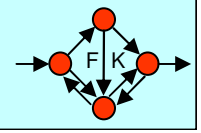


3.4.3.2 Sortieren durch Einfügen in einen Baum

Wenn man beliebige binäre Suchbäume verwendet, so können diese im schlechtesten Fall zu einer Liste entarten und daher beträgt im worst case die Zeitkomplexität $O(n^2)$.

Benutzt man aber anstelle eines beliebigen Suchbaums AVL-Bäume, so ist deren Höhe durch $1.4404 \cdot n \cdot \log(n)$ nach Satz 3.2.4.7 beschränkt. Daher ist die Anzahl der Vergleiche für das Sortieren mit Bäumen auch im schlechtesten Fall durch $1.4404 \cdot n \cdot \log(n) + O(n)$ beschränkt.

In der Praxis kann man bei der Verwendung von beliebigen Suchbäumen im Mittel mit $1.3863 \cdot n \cdot \log(n) + O(n)$, bei der Verwendung von AVL-Bäumen im Mittel mit $n \cdot \log(n) + O(n)$ rechnen (siehe Satz 3.2.2.15 und Hinweis nach Satz 3.2.4.7).

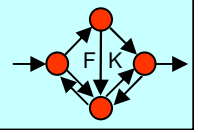


3.4.3.3 Sortieren durch Fachverteilen

Sind die Schlüssel Wörter über einem endlichen Alphabet $A = \{a_1, a_2, ..., a_s\}$, kann man die zu sortierenden Elemente zunächst bzgl. des letzten Zeichens an s Listen anfügen. Diese Listen hängt man aneineinader und fügt nun alle Elemente bzgl. des vorletzten Zeichens an s Listen an usw. Liegt die Länge jedes Schlüssels in der Größenordnung von log(n), dann ergibt sich ein $O(n \cdot \log(n))$-Sortierverfahren.

Das Anhängen an die s Listen entspricht dem Ablegen in s verschiedene Fächer, weshalb dieses Verfahren als "Fach-verteilen" bezeichnet wird.

Wir erläutern das Verfahren nur an einem Beispiel mit s=2. Die Programmierung ist nicht schwierig.



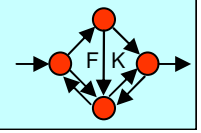
3.4.6 Streuen und Sammeln

Manchmal liegen die zu sortierenden Werte a_1 a_2 ... a_n in einem festen Intervall [UNT, OB]: UNT ≤ a_i ≤ OB.
Der Einfachheit halber nehmen wir an, die a_i seien natürliche Zahlen und es seien UNT = 0 und OB = m-1.

Bucketsort: Verteile ("streue") die n Elemente A(1..n) auf m nacheinander angeordnete Fächer bucket(0..m-1) ("Eimer" genannt, daher "bucketsort"), hole sie anschließend in der Reihenfolge der Fächer wieder heraus und lege sie im Feld A ab.

Aufwand: O(n+m) sowohl für die Zeit als auch für den Platz.

Programm zum Sortieren (fügen Sie die Listenbearbeitung selbst hinzu):



Aufgabe 7 (Fachverteilen**)**

Fachverteilen (radix exchange) ist ein Verfahren, bei dem Elemente in Fächern zwischengespeichert werden. Es setzt voraus, dass die zu sortierenden Elemente (hier Integer-Zahlen) sich aus einzelnen Ziffern (oder allgemeiner: Symbolen) zusammensetzen. Diese Symbole müssen aus einem bekannten Alphabet A entnommen sein, d. h. für jedes a_i gilt a_i ist Element von A^k, wobei *k* fest ist. Für die Zahl 123 im Dezimalsystem gilt damit A = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) und k = 3. Es wird ferner von einer lexikographischen Ordnung der Elemente in A ausgegangen. Der Fachverteilen - Algorithmus arbeitet bei einer Eingabe von *n* Elementen $a_1,.. , a_n$ folgendermaßen:

Sehen Sie |A| = *m* viele Fächer $F_1, .. , F_m$ vor, wobei jedes Fach wie ein Puffer organisiert ist. Betrachten Sie für jeden Durchlauf jeweils eine Stelle j <= k (vom der niederwertigsten bis zur höchsten Stelle).

Bearbeiten Sie die Elemente in der Reihenfolge $a_1, .. , a_n$ wie folgt: Legen Sie das aktuelle Element in das Fach F_1, falls das Symbol der gerade betrachteten Stelle j das erste Symbol des Alphabets ist.

Erzeugen Sie eine neu angeordnete Folge (die wir wieder $a_1, .., a_n$ nennen), indem die Inhalte der Fächer $F_1 , .., F_m$ in dieser Reihenfolge hintereinander abgelegt werden.

Sortieren Sie die Zahlenfolge 5, 8, 1, 6, 3, 4, 9, 10, 4, 5, 4, 7 mit Fachverteilen. Betrachten Sie hierzu die Zahlen in Binärdarstellung.

Aufgabe 8 (Quicksort)
Wenden Sie den Quicksort-Algorithmus an, um die Zahlenfolge

A = (3, 7, 9, 12, 13, 8, 6, 5, 4, 11, 10)

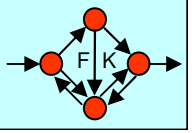
aufsteigend zu sortieren. Verwenden Sie dafür folgende Pivot-Strategien:

a) Jeweils das letzte Element der Folge wird als Pivot-Element gewählt.

b) Jeweils das erste Element der Folge wird als Pivot-Element gewählt.

c) Das mittlere Element (Folgenlänge div 2) wird als Pivot-Element gewählt.

Aufgabe 9 (Sortieren mit Suchbäumen - siehe auch Blatt 14 aus WS 03/04)
Beschreiben Sie, wie man binäre Suchbäume zum Sortieren benutzen kann.
Ein binärer Suchbaum soll durch Einfügen von Elementen aus einer Liste neu aufgebaut werden. Berechnen Sie den Aufwand des Sortierverfahrens für den besten / schlechtesten Fall.



Definition 3.4.1.5:

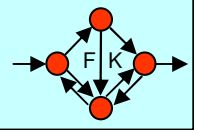
Ein Sortierverfahren *Sort* heißt **stabil**, wenn *Sort* die Reihenfolge von Elementen mit gleichem Schlüssel nicht verändert, d.h.: wenn $Sort(v_1 v_2 ... v_n) = v_{\pi(1)} v_{\pi(2)} ... v_{\pi(n)}$ die Sortierabbildung ist, dann gilt für alle $v_{\pi(i)} = v_{\pi(j)}$ mit $\pi(i) < \pi(j)$ stets $i < j$.

Sort heißt invers stabil, wenn die Reihenfolge der Elemente mit gleichem Schlüssel von *Sort* gespiegelt wird.

Wir haben bereits einige Sortierverfahren kennen gelernt:

In 1.4.4.4: Sortieren durch Austauschen benachbarter, falsch stehender Elemente (*Bubble Sort*).

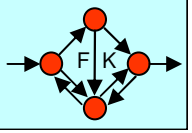
In 1.6.4.4: *Sortieren mit Bäumen,* indem die Glieder der Folge nacheinander in einen binären Suchbaum eingefügt und anschließend in inorder-Reihenfolge ausgelesen werden.



Aufgabe 10 (Stabilität)
Wenden Sie Selectionsort, Insertionsort, Bubblesort auf die unten angegebenen Zahlenfolgen an:

a) 10, 3, 9, 4, 8, 5, 7, 6
b) 12, 2, 3, 5, 7, 9, 10, 11
c) 2, 6, 3, 7, 4, 8, 5, 9

Bestimmen Sie die Anzahl der benötigten Schlüsselvergleiche und Bewegungen. Überprüfen Sie, ob die Sortierverfahren Selectionsort, Insertionsort, Bubblesort und Quicksort stabil sind (d.h. die Reihenfolge von Elementen mit gleichem Sortierschlüssel wird während des Sortierverfahrens nicht vertauscht).

Aufgabe 11 (Quicksort)
Sortieren Sie folgendes Feld mit dem Verfahren Quicksort;
geben Sie jeden Zwischenschritt an und markieren Sie jeweils das Pivotelement, sowie die Zeigerpositionen. 8, 2, 7, 16, 5, 3, 1, 17, 0, 2

Aufgabe 12 (Heapsort)
Sortieren Sie das folgende Feld mit dem Algorithmus Heapsort; geben Sie Heapaufbau und Einsinkschritte explitzit an! 4, 25, 18, 6, 5, 9, 21, 2