Kapitel 3 Sortierverfahren

Sortieren ➙ Anordnen einer gegebenen Menge von Objekten in einer bestimmten **Ordnung.**

Sortierte Folgen von Objekten bringen eine deutliche Vereinfachung für den Zugriff auf einzelne Elemente der Objektmenge. Sortieren ist daher eine grundlegende Tätigkeit, die in vielen Anwendungsbereichen angewandt wird, z.B. bei

- Telefonbüchern / Wörterbüchern / Büchereikatalogen *(Sortierung nach Buchstaben)*
- Zugfahrplänen / Flugplänen / Terminkalendern *(Sortierung nach Datum / Uhrzeit)*
- Kundenlisten / Inventarlisten *(Sortierung nach Schlüsseln)*
- LRU- (least recently used) Schlange *(Sortierung nach Zeitpunkt der letzten Nutzung)*

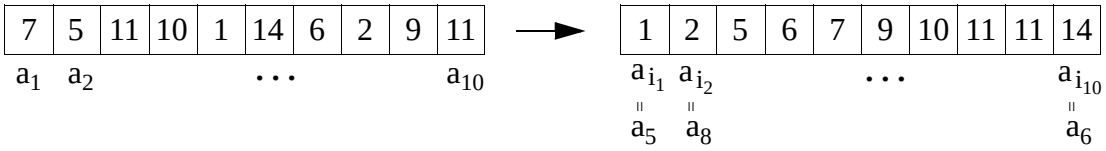
Wir betrachten die folgende allgemeine **Problemstellung**:

Gegeben: n Objekte $a_1, a_2, \ldots, a_n$ mit ihren Sortierschlüsseln $k_1, k_2, \ldots, k_n$.

Gesucht: Eine Anordnung $a_{i_1}, a_{i_2}, \ldots, a_{i_n}$ mit:

- $(i_1, i_2, \ldots, i_n)$ ist eine Permutation von $(1, 2, \ldots, n)$ und
- $k_{i_1} \le k_{i_2} \le \ldots \le k_{i_n}$ (*Sortierkriterium*).

Beispiel:



Anmerkung: Die Sortierschlüssel müssen einem Datentyp angehören, auf den das Sortierkriterium anwendbar ist, d.h. auf dem eine '<'-Relation definiert ist. Dies gilt in natürlicher Weise für atomare Datentypen wie **int**, **float** oder **char**.

Sei im folgenden die nachstehende allgemeine Klassendeklaration vorgegeben:

```
class Entry { public int key;
              ...
              }
```

```
class SortableArray {
        public Entry a [];
        // Im Konstruktor intialisieren mit new Entry [n+1];
        // Die Schlüssel sind in a[1].key bis a[n].key gespeichert
        // a[0] wird lediglich zum Vertauschen verwendet
        protected int n = a.length-1;
        protected vertausche ( int k, int l)
            { a[0]= a[k]; a[k] = a[l]; a[l] = a[0]; }
        ...
        }
```

Allgemeiner Sortieralgorithmus:

while $\exists(i,j) \;:\; (i<j) \wedge (k_i > k_j)$ **do** vertausche a_i und a_j;

➞ nicht deterministisch

➞ geeignete Kontrollstruktur (Algorithmus) nötig!

Die Vielzahl möglicher Sortierverfahren wird allgemein in zwei Kategorien eingeteilt. In Abhängigkeit davon, ob die Objekte im Hauptspeicher oder auf dem Sekundärspeicher liegen, unterscheiden wir:

- Sortieren von *Arrays* (**internes Sortieren**)
- Sortieren *sequentieller Files* (**externes Sortieren**)

Externe Sortierverfahren kommen für sehr große Datenmengen zur Anwendung, die nicht komplett in den Hauptspeicher eingelesen werden können.

internes Sortieren: alle Objekte sind bekannt

externes Sortieren:
zu jedem Zeitpunkt ist nur ein Teil der Objekte bekannt

Definition: *Stabilität* von Sortierverfahren

Ein Sortierverfahren heißt **stabil**, wenn die relative Ordnung von Elementen mit gleichen Schlüsselwerten beim Sortieren erhalten bleibt, d.h. wenn für die sortierte Folge $k_{i_1}, k_{i_2}, \ldots, k_{i_n}$ gilt: $k_{i_j} = k_{i_l}$ und $j < l \;\Rightarrow\; i_j < i_l$.

Weitere Klassifikationen von Sortierverfahren sind möglich:

- über ihr methodisches Vorgehen (Sortieren durch Einfügen oder durch Auswählen; Divide-and-Conquer - Verfahren; ...),
- über ihr Laufzeitverhalten (im schlechtesten Fall / im Durchschnitt) und
- über ihren Speicherplatzbedarf (*in situ* oder zweites Array / zusätzlicher Speicher nötig)

Für uns wesentliches Kriterium ➙ **Laufzeitverhalten**

Um unterschiedliche Verfahren miteinander zu vergleichen, zählen wir die Anzahl der bei einer Sortierung von n Elementen durchzuführenden Operationen (in Abhängigkeit von n). Die beiden wesentlichen Operationen der im folgenden betrachteten internen Sortierverfahren sind:

- **Vergleiche** von Schlüsselwerten ➙ Informationen über die vorliegende Ordnung
- **Zuweisungsoperationen**, z.B. Vertauschungsoperationen oder Transpositionen von Objekten (im allg. innerhalb eines Arrays) ➙ Herstellen der Ordnung.

Im folgenden bezeichne $C(n)$ die *Anzahl der Vergleiche* und $M(n)$ die *Anzahl der Zuweisungsoperationen*, die bei der Sortierung von n Elementen notwendig sind.

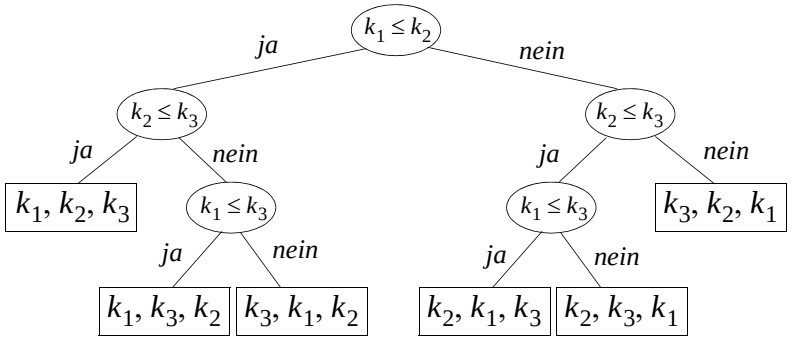
3.1 Interne Sortierverfahren

Frage: *Wie schnell kann man überhaupt sortieren?*

Beschränkung auf solche Algorithmen, die nur Schlüsselvergleiche und Transpositionen verwenden.

Gesucht: Eine untere Schranke für die Anzahl $C_{max}(n)$ von Schlüsselvergleichen, die im schlechtesten Fall notwendig sind, um n Objekte zu sortieren.

Entscheidungsbaum für 3 Schlüssel k_1, k_2, k_3 :



In einem Entscheidungsbaum ohne redundante Vergleiche entspricht jedes Blatt einer der $n!$ verschiedenen Permutationen von n Elementen.

➙ der Entscheidungsbaum für n Elemente hat eine minimale Höhe von $\lceil \log_2(n!) \rceil$ da es sich um einen binären Baum handelt

Damit gilt für einen beliebigen Sortieralgorithmus: $\boldsymbol{C_{max}(n) = \Omega(n \cdot \log n)}$, denn

$$C_{max}(n) \geq \lceil \log_2(n!) \rceil \quad \text{und}$$

$$n! \geq n \cdot (n-1) \cdot \ldots \cdot (\lceil n/2 \rceil) \geq (n/2)^{n/2}$$

$$\Rightarrow \log_2(n!) \geq n/2 \cdot \log_2(n/2) = O(n \cdot \log n) \quad .$$

$$\rightarrow C_{max}(n) \geq O(n \cdot \log n) \Leftrightarrow C_{max}(n) = \Omega(n \cdot \log n) \ .$$

Resultat: Sortierverfahren haben mindestens eine Laufzeit von $O(n \cdot \log n)$.

Da Algorithmen, die diese Laufzeit erreichen, recht komplex sind, gehen wir zunächst auf eine Reihe einfacher Sortierverfahren ein. Für kleines n (z.B. $n < 100$) bieten diese meist eine ausreichende Performanz.

3.1.1 Einfache Sortieralgorithmen

Einfache Sortieralgorithmen besitzen meist eine Laufzeit von $O(n^2)$. Wir stellen im folgenden verschiedene solche Sortieralgorithmen vor.

Sortieren durch Abzählen

Prinzip: Der *j*-te Schlüssel der sortierten Folge ist größer als *j-1* der übrigen Schlüssel. Die Position eines Schlüssels in der sortierten Folge kann damit durch Abzählen der kleineren Schlüssel ermittelt werden.

Wir benötigen ein Hilfsarray:

```
int zaehler[] = new int [n + 1];
// Die Schlüssel sind in a[1].key bis a[n].key gespeichert
```

Ziel:

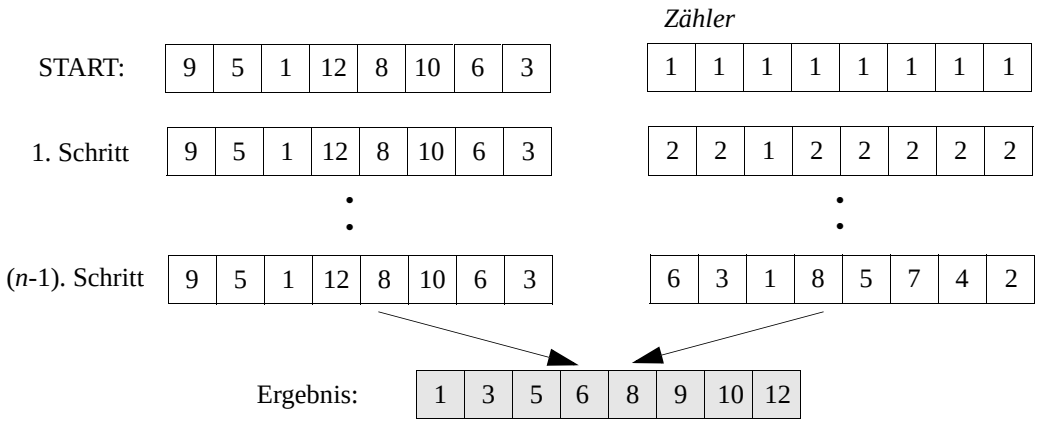
zaehler[i] = (Anzahl der Elemente a_j mit $a_j.key < a_i.key$) + 1.

Anfangs:

zaehler[] = {1, 1, ... ,1}.

Programmstück:

```
for (int i = n; i >= 2; i--) {
    for (int j = i-1; j >= 1; j--) {
        if (a[i].key < a[j].key) zaehler[j]++;
            else zaehler[i]++;
    }
}
```



Der Algorithmus ist stabil; er wird instabil, falls ‘<‘ ersetzt wird durch ‘≤’.

Für die benötigte Anzahl von Vergleichen gilt:

$$C(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n \cdot (n-1)}{2} = O(n^2) \quad .$$

Die Anzahl der Zuweisungsoperationen für den Zähler ist identisch zur Anzahl der Schlüsselvergleiche. Hinzu kommt ein linearer Aufwand, da jedes Objekt jetzt noch an die richtige Stelle plaziert werden muß.

➞ $M(n) = O(n^2)$.

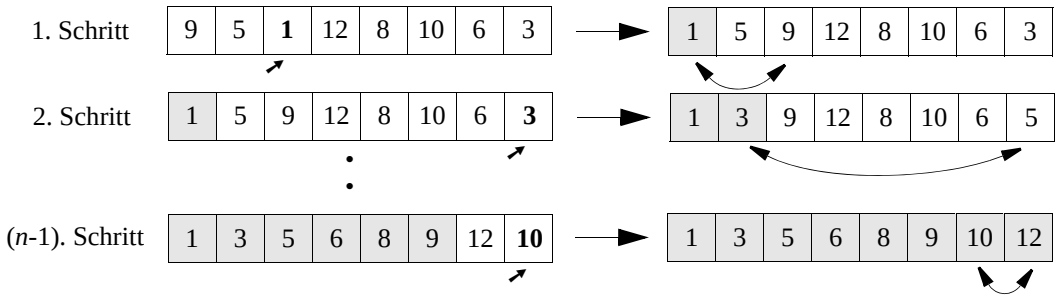
Da der Speicheraufwand von O(n) für ein Hilfsfeld groß ist, betrachten wir im folgenden eine Reihe ‘einfacher’ und stabiler Sortierverfahren, die kein zusätzliches Feld benötigen, also in situ arbeiten.

Sortieren durch direktes Auswählen (‘Selection Sort’)

Prinzip: Suche aus allen *n* Elementen das kleinste und setze es an die erste Stelle. Wiederhole dieses Verfahren für die verbleibenden *n-1*, *n-2*, ... Elemente. Nach *n-1* Durchläufen ist die Sortierung abgeschlossen.

Methode:
```
void selectionSort() {
    int minindex;
    for (int i = 1; i <= n-1; i++) {
        minindex = i;
        for (int j = i+1; j <= n; j++) // minimales Element suchen
            if (a[j].key < a[minindex].key) minindex = j;
        vertausche (i, minindex);
        }
    }
```

START: | 9 | 5 | 1 | 12 | 8 | 10 | 6 | 3 |



Für die Anzahl der Vergleichsoperationen gilt wie oben:

$$C(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n \cdot (n-1)}{2} = O(n^2) \quad .$$

Für die durchschnittliche Anzahl von Zuweisungen (‘minindex = j’) gilt (*ohne Beweis*):

$M_{\varnothing}(n) = O(n \cdot \log n)$. (Anzahl von Vertauschungen: $O(n)$)

Nähere Untersuchung des Selection_Sort:

Wesentlicher Aufwand ➙ Bestimmung des Minimums der verbleibenden Elemente.

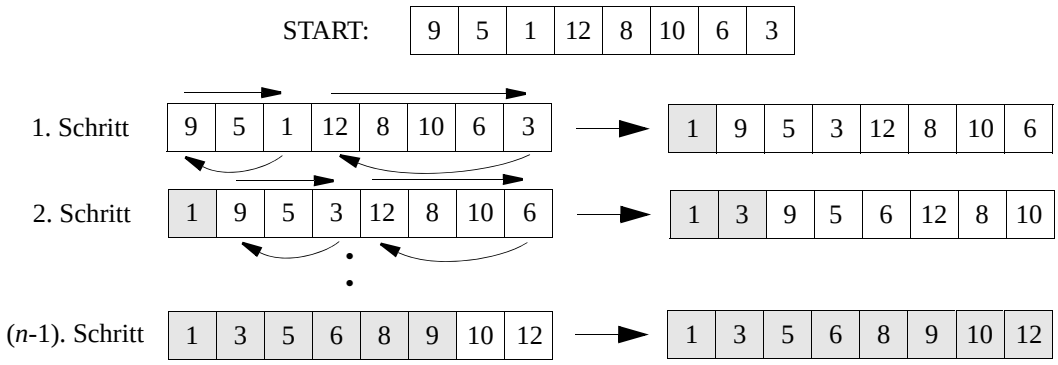
Lemma: Jeder Algorithmus zum Auffinden des Minimums von *n* Elementen, der auf Vergleichen von Paaren von Elementen basiert, benötigt mindestens *n-1* Vergleiche.

Sortieren durch direktes Austauschen (‘Bubble Sort’)

Prinzip: Das Verfahren basiert darauf, die relative Reihenfolge benachbarter Elemente zu überprüfen und diese so zu vertauschen, daß kleine Elemente an den Anfang und größere Elemente an das Ende des Arrays wandern.

Im ersten Durchlauf werden auf diese Weise die Paare (a_{n-1}, a_n) , (a_{n-2}, a_{n-1}) , ... bearbeitet. Hierdurch wandert das kleinste Element an die erste Position im Array; es wird nicht weiter betrachtet. So entsteht sukzessive eine immer längere sortierte Folge von Elementen am Anfang des Arrays und nach *n-1* Durchläufen ist die Sortierung abgeschlossen.

Methode:
```
void bubbleSort() {
    for (int i = 1; i <= n-1; i++) {
        for (int j = n; j >= i+1; j--)
            if (a[j].key < a[j-1].key) vertausche (j, j-1);
        }
    }
```



Für die benötigten Schlüsselvergleiche gilt wiederum:

$$C(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n \cdot (n-1)}{2} = O(n^2) \quad .$$

Die mittlere Anzahl von Vertauschungen beträgt:

$$M_{\varnothing}(n) = \frac{n \cdot (n-1)}{4} = O(n^2) \ .$$

Verbesserungen:

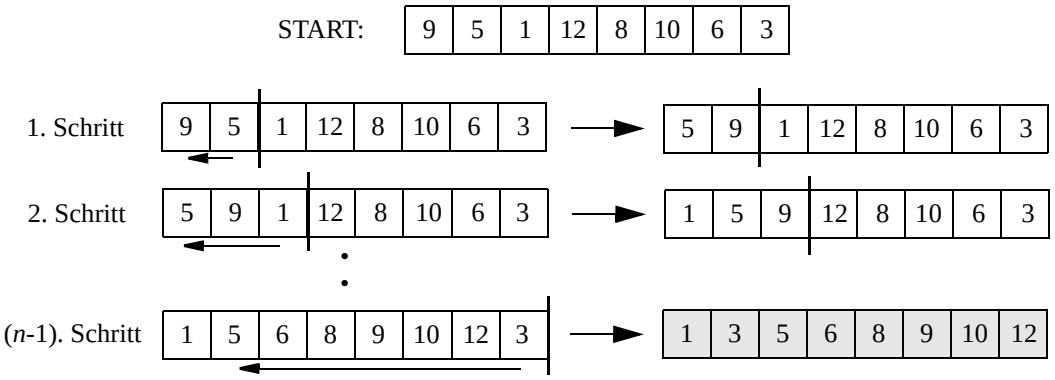
- ein Durchlauf der inneren **for**-Schleife ohne jegliche Vertauschung zweier Elemente ermöglicht einen Abbruch des Algorithmus.
- unterhalb der Position der letzten Vertauschung ist das Array bereits sortiert
  ➞ weitere Einsparung möglich durch frühzeitigen Schleifenabbruch
- Beobachtung:
  Asymmetrisches Verhalten: kleine Schlüssel springen schnell an den Anfang des Array, während große Schlüssel nur schrittweise ans Ende wandern.
  ➞ Änderung der Richtung aufeinanderfolgender Durchläufe (‘*Shaker Sort*’).

aber: die Verbesserungen sparen nur Schlüsselvergleiche, keine Vertauschungsoperationen!

Sortieren durch direktes Einfügen (‘Insertion Sort’)

Prinzip: Das Element a_1 beschreibt für sich genommen stets eine korrekt sortierte ‘Folge’. Seien nun für $j = 2, 3, \ldots$ die ersten *j-1* Elemente $a_1, a_2, \ldots, a_{j-1}$ korrekt sortiert. Im *j*-ten Durchlauf wird dann das Element a_j an die richtige Stelle der vorsortierten Folge eingefügt. Nach *n-1* Durchläufen ist die Sortierung abgeschlossen.

Methode:
```
void insertionSort() {
    Entry x = new Entry;
    for (int j = 2; j <= n; j++) {
        x = a[j]; i = j-1;
        while ((i > 0) && (x.key < a[i].key)) {
            a[i+1] = a[i];    // Transposition der ARRAY-Elemente
            i = i-1;
        }
        a[i+1] = x;
    }
}
```



Laufzeitanalyse:

- die äußere (**for**-) Schleife wird stets *n-1* mal durchlaufen.
- beim Einfügen des *j*-ten Elementes ((*j-1*)-ter Durchlauf der **for**-Schleife) werden benötigt: im worst case: *j-1* Schlüsselvergleiche und *j* Transpositionen
  im Durchschnitt: $(j-1)/2$ Schlüsselvergleiche und $(j+1)/2$ Transpositionen

$$C_{\varnothing}(n) = \sum_{j=2}^{n} \frac{j-1}{2} = \frac{1}{2}\sum_{j=2}^{n} j - \frac{n-1}{2} = \frac{n^2+n-2}{4} - \frac{2n-2}{4} = \frac{1}{4}(n^2-n) = O(n^2)$$

→

$$C_{worst}(n) = \sum_{j=2}^{n} j - 1 = \sum_{j=1}^{n-1} j = \frac{1}{2}\cdot(n^2-n) = O(n^2)$$

und ebenfalls: $O(n^2)$ Transpositionen.

Verbesserung: Beim Einfügen statt der **while**-Schleife binäres Suchen anwenden.

→ Anzahl der Schlüsselvergleiche nur noch $O(n \cdot \log n)$ im Durchschnitt sowie im schlechtesten Fall.

aber: die Anzahl der Transpositionen bleibt erhalten → Aufwand: $O(n^2)$ bleibt.

Bemerkung: Ein Sortierverfahren, das Objekte immer nur um eine Position verschiebt, hat eine durchschnittliche Laufzeit von mindestens $O(n^2)$. (*Begründung: folgendes Lemma*)

Lemma: Sei $k_1, k_2, \ldots, k_n$ eine zufällige Permutation von $\{1, 2, \ldots, n\}$. Dann gilt für die durchschnittliche Anzahl $M_{\varnothing}(n)$ der Stellen, über die die *n* Objekte bewegt werden:

$$M_{\varnothing}(n) \geq \Theta(n^2).$$

Folgerung: Sortierverfahren, die im Durchschnitt schneller als $O(n^2)$ sind, müssen Transpositionen in Sprüngen statt nur stellenweise vornehmen. (wie z.B. beim 'Selection_Sort')

3.1.2 Verfeinerung des Insertion_Sort: Shell_Sort

(Shell 1959; '*diminishing increment sort*')

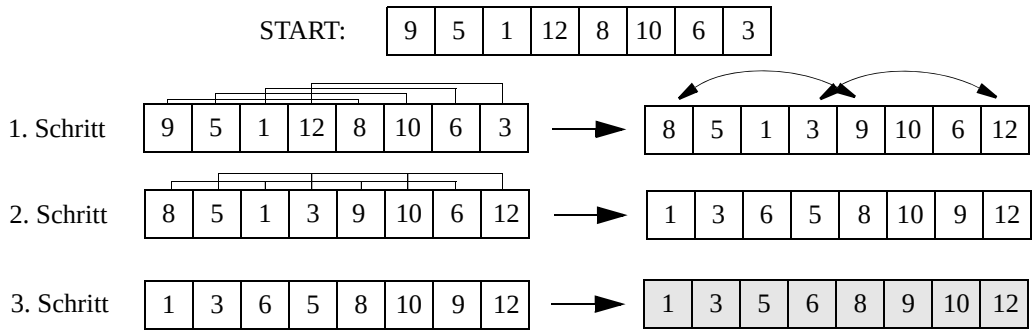
Prinzip: Sei $n = 2^k, k \in \aleph$. Das Array wird in jedem Schritt betrachtet als Menge kleinerer Gruppen von Elementen, beschrieben durch eine vorgegebene Schrittweite. Diese werden jeweils getrennt voneinander (und damit schneller) über Insertion_Sort sortiert. Als Schrittweitenfolge können beispielsweise die Werte $n/2$, $n/4$, ..., 1 gewählt werden.

Im 1. Schritt werden dann $n/2$ Gruppen mit jeweils 2 Elementen gebildet, im 2. Schritt $n/4$ Gruppen zu je 4 Elementen, usw.; jeder Schritt nutzt dabei die bereits bestehende Vorsortierung aus.

Nach $\log n$ Schritten ist schließlich die gesamte Folge sortiert. Für allgemeines *n* ergeben sich u.U. um eins größere Gruppen; die folgende Prozedur ist $\forall n \in \aleph$ korrekt.

Methode:

```
void shellSort() {
    int incr, j;
    incr = n / 2; // Distanz der Elemente eines Teilarrays
    while (incr > 0) {
        for ( int i = incr + 1; i <= n; i++) { // einzelne Teilarrays betr.
            j = i - incr;
            while (j > 0) {   // insertionSort durchführen
                if (a[j].key > a[j + incr].key) {
                    vertausche(j, j + incr);
                    if (j > incr) j = j - incr;
                        else j = 0;
                }
                else  j = 0;
            }
        }
        incr = incr / 2;
    }
}
```



Laufzeitanalyse (sei $n = 2^k$ *für ein k*):

- im 1. Schritt ist für die $n/2$ Teilarrays höchstens je 1 Vertauschung nötig.
- im 2. Schritt sind für die $n/4$ Teilarrays höchstens je 1+2 Vertauschungen nötig.
- im $\log n = k$-ten Schritt sind höchstens 1+2+...+ $n/2$ Vertauschungen nötig.

→ Gesamtlaufzeit im Mittel nahe bei: $O(n \cdot \sqrt{n}) = O(n^{1,5})$ *(ohne Beweis).*
(Laufzeit im 'worst case' offensichtlich: $O(n^2)$)

Bemerkungen

- Schrittweiten, die keine Potenzen von 2 sind, liefern in der Regel bessere Ergebnisse.

Experimentell ermittelt schlägt Knuth folgende (umgekehrte) Schrittweitenfolge vor:
1, 4, 13, 40, 121, ... bzw. allgemein: $h_1 = 1$, $h_{s+1} = 3 \cdot h_s + 1$;
und beginne mit der Schrittweite $\bar{s}$, für die gilt: $h_{\bar{s}+2} \geq n$.

➞ Laufzeit: $O(n^{1,25})$

• Unabhängig von der Schrittweite ➞ Laufzeit nicht besser als $O(n^{1,2})$ (> $O(n \cdot \log n)$).

3.1.3 Divide-and-Conquer-Methoden

Die einfachen Sortieralgorithmen reduzieren die Größe des noch zu sortierenden Arrays in jedem Schritt lediglich um eins. Ziel der Divide-and-Conquer Sortieralgorithmen ist es, das zu sortierende Array in jedem Schritt zu halbieren. Bei Erreichung dieses Ziels erhalten wir Verfahren mit einer Laufzeit von $O(n \cdot \log n)$.

Allgemeines Paradigma:

allgemeiner Divide-and-Conquer-Algorithmus:

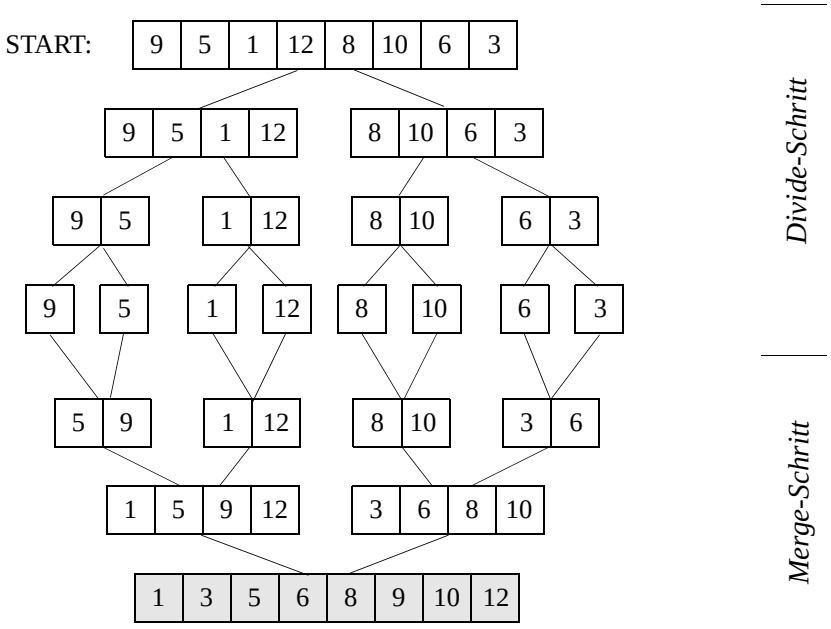
ALGORITHMUS Divide-and-Conquer_Sort;
IF Objektmenge klein genug
THEN löse das Problem direkt
ELSE
Divide: Zerlege die Menge in Teilmengen möglichst gleicher Größe;
Conquer: Löse das Problem für jede der Teilmengen;
Merge: Berechne aus den Teillösungen die Gesamtlösung
END;

MergeSort

Prinzip: Das Array wird zunächst rekursiv bis auf Elementebene zerlegt. Jedes der Elemente ist für sich genommen trivialerweise sortiert.
Im Anschluß an diese Zerlegung werden dann gemäß der Zerlegungshierarchie jeweils zwei sortierte Teilarrays derart verschmolzen (*Merge*), daß das Ergebnis wiederum sortiert ist. Nach dem letzten Merge-Schritt ist das gesamte Array sortiert.

(trivialer *Divide*-Schritt ($O(1)$); 'Arbeit' im *Merge*-Schritt ($O(n)$))

ALGORITHMUS Mergesort(S)
IF |S| = 1
THEN RETURN S
ELSE
Divide: S1 := $a_1, \ldots, a_{\lfloor n/2 \rfloor}$; S2 := $a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n$;
Conquer: S1' := Mergesort(S1); S2' := Mergesort(S2);
Merge: **RETURN** Merge(S1', S2')
END;



Laufzeitanalyse (gilt im wesentlichen für alle *Divide-and-Conquer*-Verfahren):

Die Laufzeit $T(n)$ für den ersten Rekursionsschritt für n Elemente ergibt sich aus:

- der Laufzeit des *Divide*-Schrittes
  - ➙ $O(1)$ (Aufteilen des Arrays ➙ konstante Zeit)
- der Laufzeit des *Conquer*-Schrittes
  - ➙ $2 \cdot T(n/2)$ (n Elemente ➙ $2 \cdot n/2$ Elemente)
- der Laufzeit des *Merge*-Schrittes
  - ➙ $O(n)$ (jedes Element muß betrachtet werden)

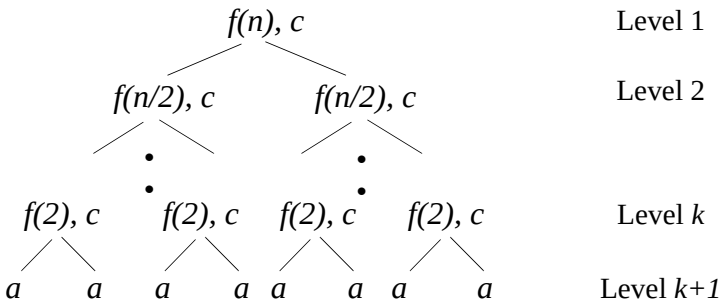
Es ergibt sich somit folgende Rekursionsgleichung für den Zeitbedarf $T(n)$ des MergeSort:

$$T(n) = \begin{cases} O(1) & \text{falls } n = 1 \\ O(1) + 2 \cdot T(n/2) + O(n) & \text{falls } n > 1 \end{cases}$$

oder anders ausgedrückt (mit Konstanten a und c und einer (mindestens) linear wachsenden Funktion f.):

$$T(n) = \begin{cases} a & \text{falls } n = 1 \\ c + 2 \cdot T(n/2) + f(n) & \text{falls } n > 1 \end{cases}$$

Betrachten wir zur Veranschaulichung den Baum der Kosten dieser Rekursionshierarchie (für $n = 2^k$):



Der Aufwand für alle *c*'s und *a*'s ist: $(n-1)\cdot c + n \cdot a = O(n)$.

Da f (mindestens) linear wächst, gilt: $i \cdot f(n/i) \le f(n)$.

Hieraus ergibt sich die folgende Abschätzung, wobei $k = \log n$:

Level 1: $f(n)$
Level 2: $2 \cdot f(n/2) \le f(n)$
...
Level *k-1*: $n/4 \cdot f(4) \le f(n)$
Level *k*: $n/2 \cdot f(2) \le f(n)$,

➞ $T(n) \le f(n) \cdot \log n = O(n \cdot \log n)$.

Allgemeines Ergebnis

Jedes *Divide-and-Conquer*-Sortierverfahren, dessen *Divide*- und *Merge*-Schritt in $O(n)$ durchgeführt werden können und das eine balancierte Unterteilung des Problems garantiert, besitzt eine Laufzeit von $O(n \cdot \log n)$.

Quicksort (Hoare 1962)

Prinzip: Ein (prinzipiell beliebiger) Schlüssel *x* wird aus dem Array ausgewählt. Das Array wird anschließend in Schlüssel $\ge x$ und Schlüssel $< x$ zerlegt (*Divide*-Schritt). Die beiden resultierenden Teilarrays werden rekursiv bis auf Elementebene in gleicher Weise behandelt (*Conquer*). Durch entsprechende Speicherung der Teilarrays innerhalb des Arrays entfällt der *Merge*-Schritt.

Arbeit im *Divide*-Schritt ($O(n)$); trivialer *Merge*-Schritt ($O(1)$)

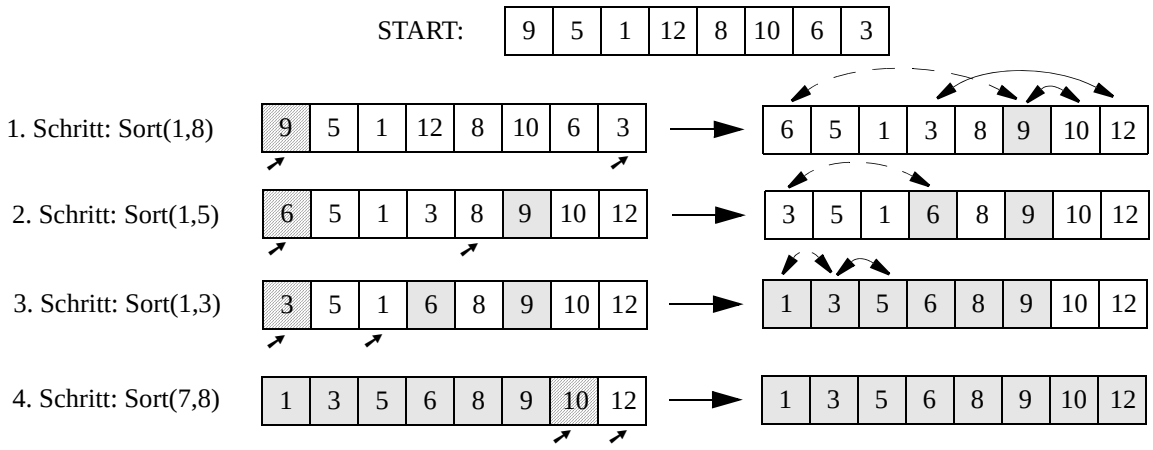
allgemeiner Algorithmus:

ALGORITHM Quicksort(S)
IF |S| = 1
THEN RETURN S
ELSE
Divide: Wähle irgendeinen Schlüsselwert x = s_j.key aus S aus. Berechne eine Teilfolge S1 aus S mit den Elementen, deren Schlüsselwert < x ist und eine Teilfolge S2 mit Elementen ≥ x.
Conquer: S1' = Quicksort(S1); S2' = Quicksort(S2);
Merge: **RETURN** Concat(S1', S2')
END;

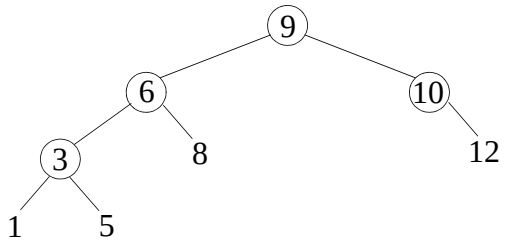
Methode:

```
void quickSort() {
    sort(1, n);
}

void sort ( int li, int r) {
    int i = li + 1;
    int j = r;
    Entry x = a[li]; // hier: Auswahl des linken Randelementes
    boolean flag = false;
    do { // Aufteilung in Elemente > bzw. < x
            while ((a[i].key < x.key) && (i < r)) i = i+1;
            while  (a[j].key > x.key)   j = j-1;
            if (i < j) {
                    vertausche(i, j);
                    i = i+1; j = j-1;
                    }
            else flag = true;
    } while (!flag);
    vertausche(li, j); // x an richtige Stelle bringen
    if  (li < j-1)   sort(li, j-1);
    if  (j+1 < r)   sort(j+1, r);
}
```



Rekursive Unterteilung des Array bezüglich des ausgewählten Elementes
➞ binärer Baum über den Arrayelementen.



Höhe des Baumes ➞ Rekursionstiefe des Verfahrens.
erwünscht: balancierter binärer Baum ➞ Rekursionstiefe: $O(\log n)$.

Laufzeitanalyse:

Wir untersuchen zunächst den Zerlegungsschritt (*Divide*-Schritt):

Ausgangspunkt: angenommen, das k-te Element der Sortierreihenfolge
wird für den Zerlegungsschritt gewählt und dann an Position k positioniert.

- im Zerlegungsprozeß wird jedes Element betrachtet ➞ n Vergleiche
- die Anzahl der Vertauschungsoperationen ist abhängig von k:

➞ (Anzahl der Elemente < k) • (Wahrscheinlichkeit für eine Vertauschung)
(= WS, daß das Element im rechten Teil war)

$$= (k-1) \cdot \frac{n-(k-1)}{n} \quad .$$

➞ Mittelwert $\overline{M(n)}$ der Vertauschungsoperationen über alle $k = 1, 2, \ldots, n$
für den ersten Zerlegungsschritt:

$$\overline{M(n)} = \frac{1}{n} \cdot \sum_{k=1}^{n} (k-1) \cdot \frac{n-(k-1)}{n} = \frac{1}{n^2} \cdot \sum_{k=0}^{n-1} k \cdot (n-k) =$$

$$= \frac{n \cdot (n-1)}{2 \cdot n} - \frac{2 \cdot n^2 - 3 \cdot n + 1}{6 \cdot n} = \frac{n-1/n}{6} = O(n)$$

➞ der *Divide*-Schritt benötigt stets $O(n)$ Operationen.

Gesamtzeitbedarf:

Der günstigste Fall:

➞ Das Array wird jeweils in zwei Teilarrays etwa gleicher Größe geteilt.
Dann folgt für den Gesamtzeitbedarf:

$$T_{best}(n) = c \cdot n + 2T(n/2) = c \cdot n + 2(c \cdot n/2 + 2T(n/4)) =$$
$$= 2cn + 4T(n/4) = \ldots = \log n \cdot c \cdot n + n \cdot T(1) = O(n \cdot \log n) \quad .$$

Der durchschnittliche Fall:

➞ Das Array wird jeweils bezüglich eines zufällig ausgewählten Elements zerlegt.
Nach Analysen: nur um den Faktor $2 \cdot \ln(2) \approx 1{,}39$ (ca. 40%) schlechter.

➞ $T_{\varnothing}(n) = O(n \cdot \log n)$.

Der schlechteste Fall:

➞ Degenerierung (lineare Liste); stets wird das größte / kleinste Element gewählt.
(für die obige Implementierung im Falle eines bereits sortierten Elementes)

$$T_{worst}(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n \cdot (n-1)}{2} = O(n^2) \quad .$$

Beachte hierbei auch die hohe Rekursionstiefe n.

Bemerkungen zu Quicksort

- Austauschen von Schlüsseln über große Distanzen ➞ schnell ‘nahezu’ sortiert.
- wie alle komplexeren Verfahren ist Quicksort schlecht für kleine n
  ➞ Verbesserung durch direktes Sortieren kleiner Teilarrays (z.B. der Größe < 10).
- Methoden, um die Wahrscheinlichkeit des schlechtesten Falls zu vermindern:
  ➞ Zerlegungselement x als Median aus 3 oder 5 (zufälligen) Elementen bestimmen
  ➞ ‘*Clever Quicksort*’.
  dennoch: Quicksort bleibt immer eine Art ‘Glücksspiel’, denn es bleibt:

  $$T_{worst}(n) = O(n^2)$$

- Quicksort war lange Zeit das im Durchschnittsverhalten beste bekannte Sortierverfahren.

3.1.4 Sortieren mit Hilfe von Bäumen

Sortieren durch Auswählen

➞ *n*-maliges Extrahieren des Minimums/Maximums einer Folge von *n* Schlüsseln

➞ Verwenden einer geeigneten Baumstruktur, die diese Operation effizient unterstützt

Heapsort (Williams 1964)

Definition: *Heap*

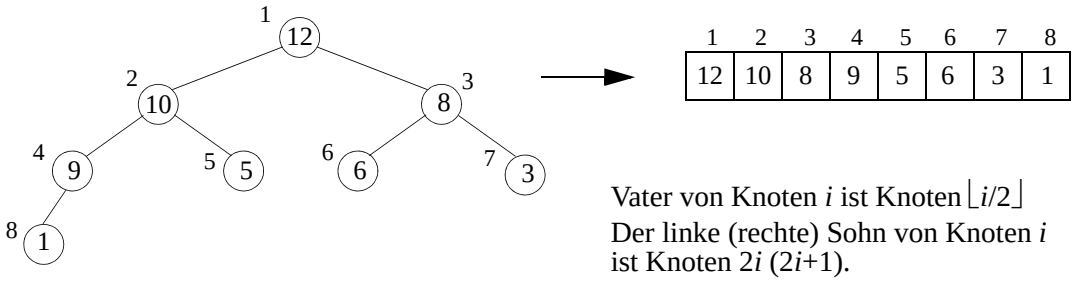
Sei $K = \{k_1, k_2, \ldots, k_n\}$ eine Menge von Schlüsseln und *T* ein binärer Baum, dessen Knoten die Schlüssel aus *K* speichern.

T ist genau dann ein *Heap*, wenn gilt:

- *T* ist vollständig ausgeglichen und
- der Schlüssel in jedem Knoten ist größer oder gleich den Schlüsseln in den Söhnen des Knotens (*partielle Ordnung*).

Ein Heap ist auf sehr einfache Art mit Hilfe eines sequentiellen Arrays implementierbar.

Beispiel:



Vater von Knoten *i* ist Knoten $\lfloor i/2 \rfloor$
Der linke (rechte) Sohn von Knoten *i* ist Knoten $2i$ $(2i+1)$.

Eine Folge von Schlüsseln $k_1, k_2, \ldots, k_n$ bildet einen *sequentiellen Heap*, wenn gilt:

$$k_{\lfloor j/2 \rfloor} \geq k_j \quad \text{für} \quad 1 \leq \lfloor j/2 \rfloor < j \leq n \ .$$

In diesem Falle gilt: $k_1 = \max_{1 \leq i \leq n} (k_i)$, d.h. das Maximum steht in der Wurzel.

Weiterhin erfüllen alle Blätter $k_{\lfloor n/2 \rfloor + 1}, \ldots, k_n$ für sich genommen bereits die Heap-Eigenschaft.

Prinzip: Die zu sortierende Schlüsselfolge wird in einem Vorbereitungsschritt in eine sequentielle Heap-Struktur überführt.
Aus dieser Struktur wird in jedem Schritt das Maximum (die Wurzel) entnommen und der Rest wiederum in Heap-Struktur gebracht. Eine sukzessive *n*-malige Entnahme des Maximums führt zur Sortierung der Folge.

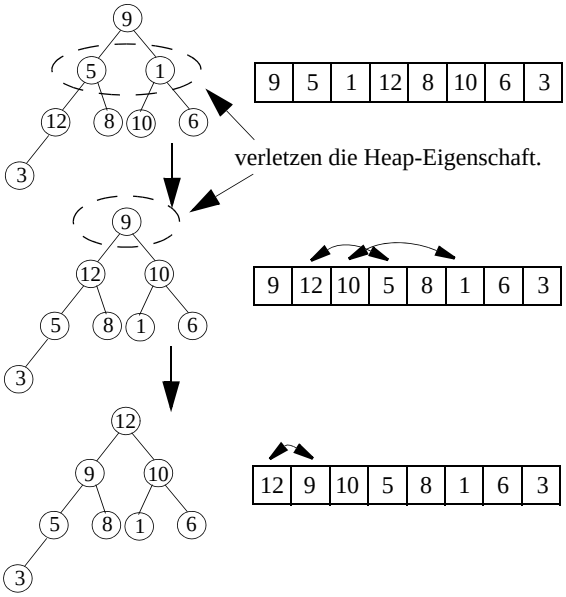
Aufbau der Heap-Struktur:

Ausgangspunkt: unsortiertes Array.

- ➞ die *n* DIV 2 Elemente in den Blättern erfüllen die Heap-Eigenschaft.
- ➞ ‘Absenken’ aller inneren Schlüssel, die diese Eigenschaft verletzen durch Vertauschen des Schlüssels mit dem größeren seiner Söhne. (*sukzessive über alle Level*)

Ergebnis: ‘*Anfangsheap*’.

- ➞ der maximale Schlüssel befindet sich in der Wurzel.

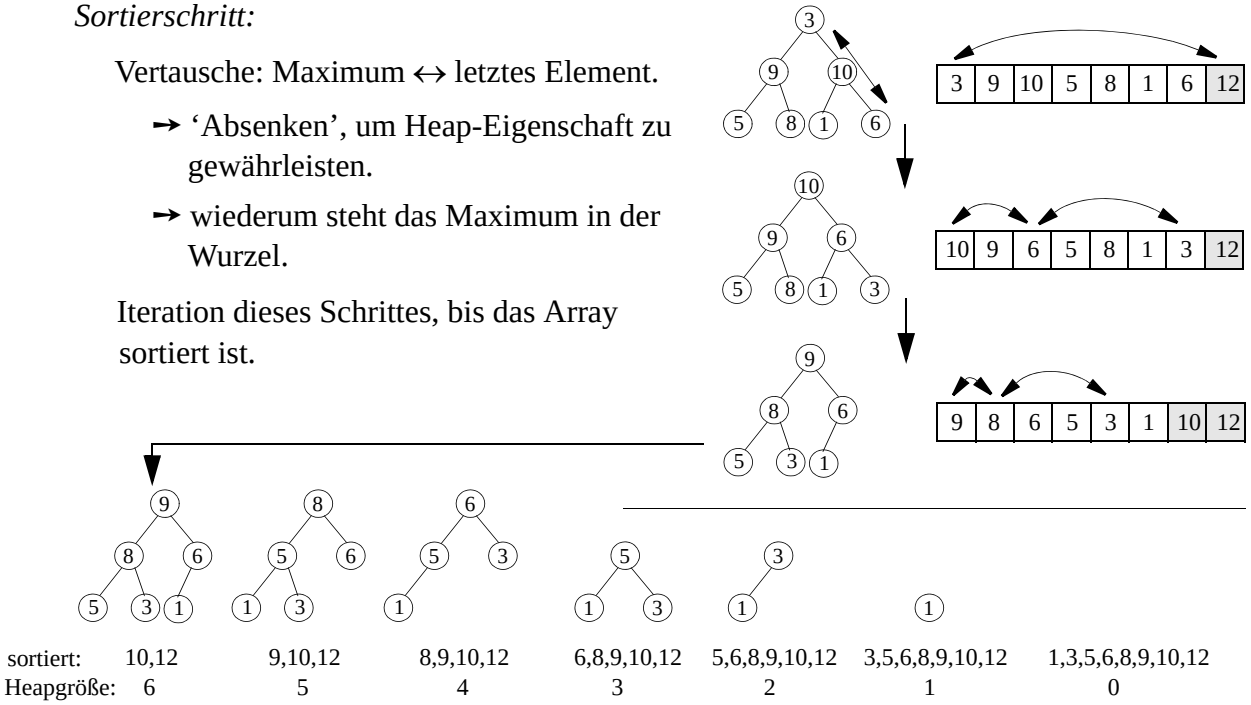


Sortierschritt:

Vertausche: Maximum ↔ letztes Element.

- ➞ ‘Absenken’, um Heap-Eigenschaft zu gewährleisten.
- ➞ wiederum steht das Maximum in der Wurzel.

Iteration dieses Schrittes, bis das Array sortiert ist.



Die folgende Methode ‘*Heapsort*’ basiert auf einer lokalen Hilfsprozedur ‘*korrigiere*’. Diese läßt das Element a[*li*] in das als Teilheap vorausgesetzte Teilarray a[*li*+1],...,a[*r*] an die richtige Stelle einsinken, so daß a[*li*],...,a[*r*] einen Heap bildet.

Methode:

```
void heapSort() {
    for (int i = n/2; i >= 1; i--) korrigiere (i,n);// Aufbau des Heap
    for (int i = n; i >= 2; i--) {                  // Sortierschritte
        vertausche (i, 1);
        korrigiere (1, i-1);
    }
}

void korrigiere ( int li, int r) {
    int i, j;
    Entry x;
    i = li;  x = a[li];    // abzusenkendes Element in der Wurzel
    if ((2*li) <= r) { // rechte Grenze nicht überschritten
        j = 2*i; // a[j] ist linker Sohn von a[i]
        do { // wiederholtes Absenken
            if (j < r)  // suche größeren der beiden Söhne
                if (a[j].key < a[j+1].key) j=j+1;
            // nun: a[j] ist größerer Sohn
            if (x.key < a[j].key) {
                vertausche(i,j);
                i=j; j=2*i;    // Absenken
                }
            else  j=r+1;  // halte an: Heap-Bedingung erfüllt
        } while (j <=r);
    }
}
```

Laufzeitanalyse:

Der Algorithmus unterteilt sich in zwei getrennte Schritte:

1) Aufbau des 'Anfangsheap':

➙ '*korrigiere*' wird einmal für jeden Knoten aufgerufen, der mind. einen Sohn hat. Für die Höhe *h* eines vollständig ausgeglichenen binären Baumes gilt:

$h = \lceil \log_2(n+1) \rceil$.

Für *n* gilt: $2^{h-1} \le n < 2^h$.

In einem derartigen Baum besitzt Level i genau 2^{i-1} Knoten für $1 \le i \le h-1$. Für jeden dieser Knoten muß der darin enthaltene Schlüssel maximal bis in einen Blattknoten wandern, d.h. einschließlich des Knotens selbst *h-(i-1)* Knoten durchlaufen.

$$T_{worst}^{Aufbau}(n) \le \sum_{i=1}^{h-1} 2^{i-1}(h-(i-1)) = \sum_{i=0}^{h-2} 2^{i}(h-i) = \sum_{i=2}^{h} (2^{h-i} \cdot i) =$$

$$= \sum_{i=2}^{h} \left(\frac{2^{h-1}}{2^{i-1}} \cdot i \right) \le n \sum_{i=2}^{h} \frac{i}{2^{i-1}} = O(n)$$

Das letzte"=" gilt, da $\sum_{i=2}^{h} \frac{i}{2^{i-1}}$ für $h \to \infty$ konvergiert.

2) Sortierung durch sukzessive Entnahme des Maximums:

➞ (n-1)-mal muß jeweils ein Schlüssel durch '*korrigiere*' abgesenkt werden.
Jedes Absenken ist auf einen Pfad von der Wurzel zu einem Blatt beschränkt.

➞ $T_{worst}^{Sortierung}(n) \le (n-1) \cdot \lceil \log_2(n-1+1) \rceil = O(n \cdot \log n)$.

Schlußfolgerung

Die Laufzeit von Heapsort im schlechtesten Fall ist also $O(n \cdot \log n)$ und damit optimal.

Bemerkungen:

- Betrachten wir die durchschnittliche Laufzeit von Heapsort, so fällt auf, daß grob 50% der Knoten eines Heaps auf der untersten Ebene, 25% auf der vorletzten Ebene, ..., liegen.
  ➞ Wahrscheinlichkeit eines absinkenden Elementes, 'weit unten' zu landen, ist groß.
  ➞ durchschnittlicher Zeitbedarf nahe bei $2 \cdot n \cdot \log n + O(n)$.
  ➞ schlechtes Durchschnittsverhalten im Vergleich zu Quicksort ($1{,}39 \cdot n \cdot \log n + O(n)$).

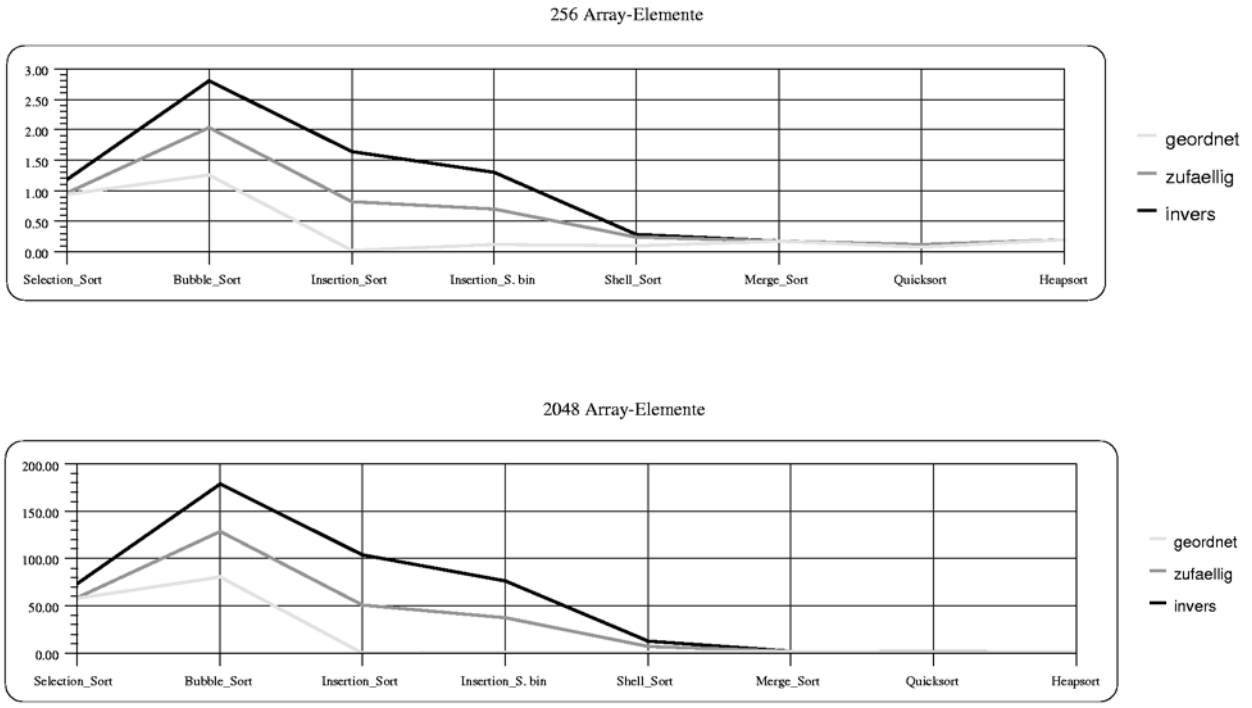
- Verbesserung: **Bottom-Up-Heapsort** (Wegener 1990)
  bisher: Beim Absenken eines Schlüssels sind jeweils zwei Vergleiche notwendig:
  'Welcher Sohn ist größer?' und 'Ist dieser größer als der abzusenkende Schlüssel?'
  ➞ die zweite Frage wird meist mit 'ja' beantwortet.
  jetzt: Betrachte nur noch die erste Frage ➞ *Zielpfad* von der Wurzel zu einem Blatt.
  'Von unten' wird nun die Einfügestelle des abgesenkten Schlüssels gesucht (meist nur 1 oder 2 Vergleiche); übrige Elemente auf dem Pfad werden nach oben geschoben.
  ➞ Zahl der Vergleiche wird reduziert.
  ➞ Bottom-Up-Heapsort ist das beste bekannte Sortierverfahren für große n ($n > 16000$).

- Da der Heap-Aufbau nur $O(n)$ Zeit benötigt, kann Heapsort dazu benutzt werden, die k größten (kleinsten) Elemente einer Schlüsselmenge in $O(n + k \cdot \log n)$ Zeit zu finden. Ist $k \le n / (\log n)$, so kann dieses Problem in $O(n)$ Zeit gelöst werden.

3.1.5 Experimentell ermittelte Laufzeit von Sortierverfahren

Wirth stellt einen experimentellen Vergleich der Laufzeit verschiedener Sortierverfahren vor.

Die folgende Grafik gibt die von ihm ermittelten Ausführungszeiten der entsprechenden Modula-Programme in Sekunden wieder:



	Anzahl der Elemente = 256			Anzahl der Elemente = 2048		
	geordnet	zufällig	invers	geordnet	zufällig	invers
Selection_Sort	0,94	0,96	1,18	58,18	58,34	73,46
Bubble_Sort	1,26	2,04	2,80	80,18	128,84	178,66
Insertion_Sort	0,02	0,82	1,64	0,22	50,74	103,80
Insertion_Sort (binär)	0,12	0,70	1,30	1,16	37,66	76,06
Shell_Sort	0,10	0,24	0,28	0,80	7,08	12,34
Merge_Sort	0,18	0,18	0,18	1,98	2,06	1,98
Quicksort	0,08	0,12	0,08	0,72	1,22	0,76
Heapsort	0,20	0,20	0,20	2,32	2,22	2,12

Einen Leistungsvergleich zwischen seinem Bottom-Up-Heapsort und dem Median-basierten Clever Quicksort hat Wegener durchgeführt. Die folgende Tabelle veranschaulicht die relative Anzahl der notwendigen Vergleiche in Abhängigkeit von der Arraygröße n:

n	Clever Quicksort	Bottom-Up-Heapsort	Verhältnis
100	5,74	6,94	0,827
200	6,88	7,99	0,861
500	8,42	9,31	0,904
1000	9,60	10,31	0,931
2000	10,78	11,31	0,953
5000	12,35	12,66	0,976
10000	13,54	13,66	0,991
20000	14,72	14,67	1,003
30000	15,42	15,24	1,012

Folgerungen:

- Clever Quicksort ist schneller für kleine n.
- Die Leistungsfähigkeit von Bottom-Up-Heapsort nimmt für wachsendes n zu.
- Ab ca. $n = 16000$ ist Bottom-Up-Heapsort besser als Clever Quicksort.
- Für weiter wachsendes n nimmt dessen Leistungsvorteil weiter zu
  ➞ Bottom-Up-Heapsort wird ca. 15 % besser für $n = 10^{20}$.

3.1.6 Sortieren in linearer Zeit

Aussage: Sortieren auf Basis von Schlüsselvergleichen benötigt im schlechtesten Fall mindestens $O(n \cdot \log n)$ Operationen.

aber: Wenn die Schlüsselwerte bestimmten Einschränkungen unterliegen und auch andere Operationen als Vergleiche angewendet werden können, dann ist es möglich, schneller als in $O(n \cdot \log n)$ Zeit (z.B. in $O(n)$ Zeit) zu sortieren.

Wir werden im folgenden z.B. annehmen

- die auftretenden Schlüssel sind a-priori bekannt
- der Wertebereich der Schlüssel ist (bekannt und) relativ klein

Beispiel:

Es sei bekannt, daß die Schlüsselfolge von $a_1, a_2, \ldots, a_n$ eine Permutation von $\{1, 2, \ldots, n\}$ darstellt. Dann kann die folgende Prozedur '*Sort*' zur Sortierung herangezogen werden:

Methode:
```
void sort() {
    for (int i = 1; i <= n; i++) {
        while (a[i].key != i)
            vertausche(i, a[i].key);
        }
    }
```

Diese Methode '*sort()*' benötigt nur $O(n)$ Zeit für die Sortierung von n Elementen.

Eine Verallgemeinerung dieses Verfahrens stellt das sogenannte Bucket_Sort dar:

Bucket_Sort

Sei der Wertebereich der Schlüssel gegeben als $[0, 1, \ldots, m-1]$, Duplikate der Schlüssel seien erlaubt. Gegeben sei eine Folge von Behältern (buckets) $B_0, B_1, \ldots, B_{m-1}$. Jedes Bucket kann eine Liste von Elementen aufnehmen. Wir benutzen hierzu die gleiche Array-Datenstruktur wie bei offenen Hashverfahren, d.h. mit Überlauflisten. Neue Elemente werden immer am Ende der Überlaufliste eingefügt.

Dann sortiert folgender Algorithmus die Schlüsselfolge $a_1, a_2, \ldots, a_n$:

```
ALGORITHMUS Bucket_Sort; (* das Feld a sei globale Variable *)
BEGIN
    FOR i := 1TO n DO
        füge a_i in das Bucket B_{a_i.key} ein END;
    FOR i := 0 TO m-1 DO
        schreibe die Elemente von B_i in die Ergebnisfolge END
END Bucket_Sort;
```

Dieser Algorithmus wird auch als "Sortieren durch Fachverteilung" bezeichnet.

Laufzeitanalyse:

- Einfügen eines Elementes in ein Bucket benötigt $O(1)$ Zeit
  (z.B. lineare Liste mit Zeiger auf letztes Element)

➞ Bucket_Sort benötigt $O(n + m)$ Zeit

falls $m = O(n)$ ➞ $O(n)$ Zeit

Bemerkung: Das Zuordnen eines Schlüssels a_i zum 'richtigen' Bucket $B_{a_i.key}$ stellt eine m-wertige Vergleichsoperation dar. Diese ist gleichwertig zu $\log_2(m)$ binären Vergleichsoperationen. Somit steht dieser Algorithmus nicht im Widerspruch zur 'theoretischen Komplexität' von Sortierverfahren von $O(n \cdot \log n)$.

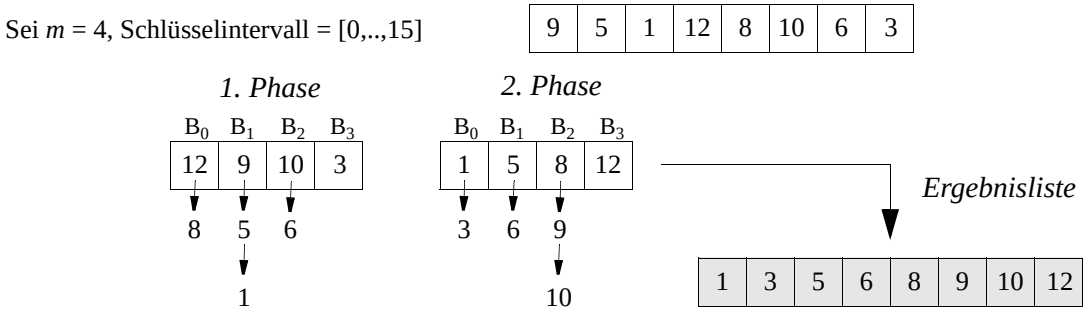
Häufig haben die Schlüssel einen sehr großen Wertebereich ➞ zu viele Buckets notwendig!
➞ **Bucket_Sort in mehreren Phasen**.
(k Phasen, falls der Wertebereich der Schlüssel gleich $[0, 1, \ldots, m^k - 1]$)

Beispiel: Sei $k = 2$ und der Wertebereich der Schlüssel gleich $[0, 1, \ldots, m^2 - 1]$.
Dann werden die beiden folgenden Sortierphasen durchgeführt:

1.Phase: Sortierung *innerhalb* der Buckets
Bucket_Sort mit m Buckets, wobei a_i in das Bucket $B_{a_i.key \text{ MOD } m}$ eingefügt wird.

2.Phase: Sortierung *zwischen* den Buckets
Durchläuft die Buckets der 1.Phase und hängt a_i jeweils an die Liste von Bucket $B_{a_i.key \text{ DIV } m}$ an.

Die Ergebnisse der 2. Phase werden schließlich in die Ergebnisliste geschrieben.



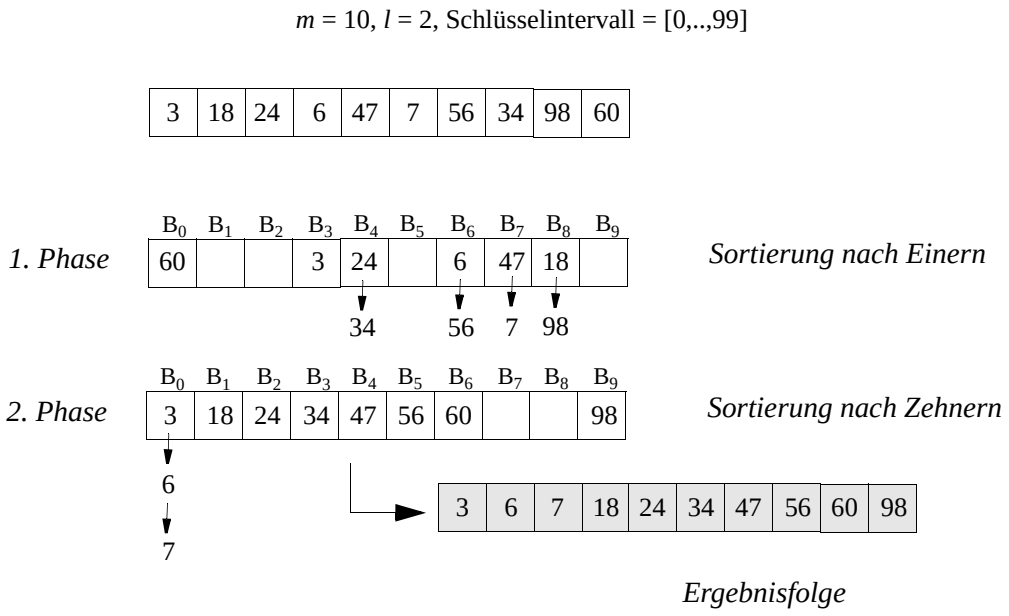
Nimmt man k als Konstante an, so arbeitet dieses Verfahren in $O(n)$ Zeit, falls das Anhängen an eine Liste in $O(1)$ Zeit erfolgt.

Im folgenden stellen wir ein *allgemeineres Verfahren (Radix_Sort)* vor, bei dem die Schlüsselwerte als Ziffernfolge zur Basis m aufgefaßt werden.

Radix_Sort

Seien die Schlüsselwerte Zeichenfolgen über einem Alphabet von *m* Buchstaben mit gleicher Länge *l*, die als Zahlen zur Basis *m* interpretiert werden. Der Wertebereich der Schlüssel entspricht also $[0, 1, \ldots, m^l - 1]$. Gegeben sei eine Folge von Buckets $B_0, B_1, \ldots, B_{m-1}$, wobei jedes Bucket eine Liste von Elementen aufnehmen kann. Neue Elemente werden immer "am Ende" des Buckets eingefügt.
Es werden *l* Phasen durchgeführt. Jede Phase hängt ab von der jeweils betrachteten Ziffer an Position *j* der *m*-adischen Schlüssel, wobei *j* mit der niedrigstwertigen Position 0 beginnt und alle Positionen bis *l*-1 durchläuft.

In Phase *j* werden die Datensätze so auf die Buckets verteilt, daß B_i alle Datensätze enthält, deren Schlüssel an *j*-ter Position das *i*-te Zeichen besitzt. Bei dieser Verteilung bleibt die relative Anordnung der Datensätze innerhalb eines jeden Buckets, die aus den früheren Phasen stammt, unverändert erhalten. Zum Schluß wird die Ergebnisfolge mit einem Durchlauf der Buckets in aufsteigender Folge ihrer Nummern ($B_0, B_1, \ldots, B_{m-1}$) generiert.



Laufzeitanalyse:

- Falls *l* als konstant angesehen werden kann und $m < n$ gilt, ist die Laufzeit von Radix_Sort O(*n*).
- Wenn alle *n* Schlüssel verschieden sind, gilt $l \geq \lceil \log_m n \rceil$. Solange $l = c \cdot \lceil \log_m n \rceil$ mit einer kleinen Konstanten *c*, besitzt Radix_Sort eine Laufzeit von $O(n \cdot \log n)$.
- Diese Laufzeiten gelten auch im schlechtesten Fall.