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 Verfahren 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 $O(n \cdot \log n)$.

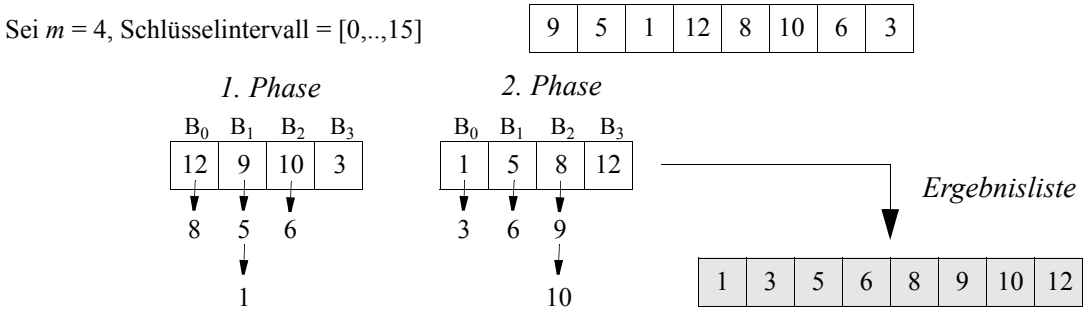
Häufig haben die Schlüssel einen sehr großen Wertebereich ➞ zu viele Buckets notwendig!
➞ **Bucket_Sort in mehreren Phasen**.
(k Phasen, falls der Wertebereich der Schlüssel gleich $[0, 1, \ldots, m^k - 1]$)

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.



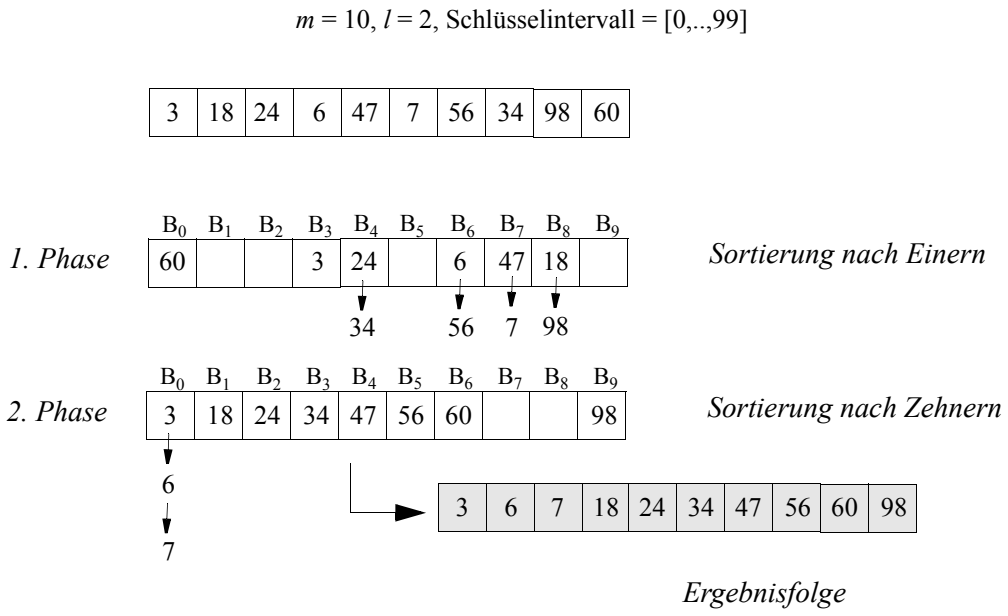
Nimmt man k als Konstante an, so arbeitet dieses Verfahren in $O(n)$ Zeit, falls das Anhängen an eine Liste in $O(1)$ Zeit erfolgt.

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.