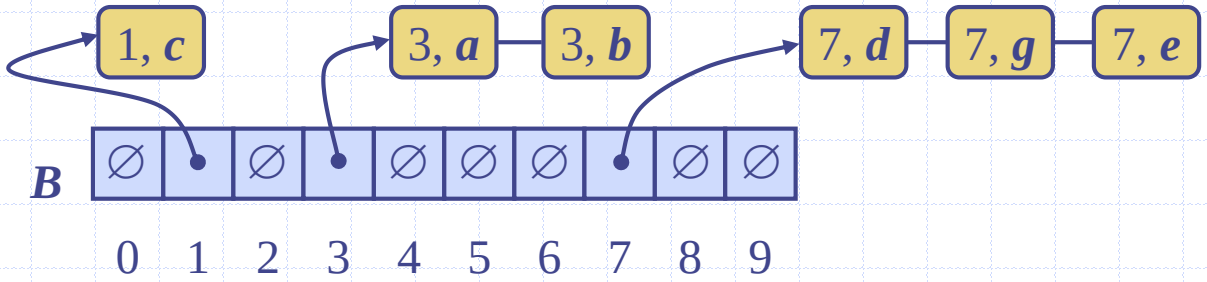
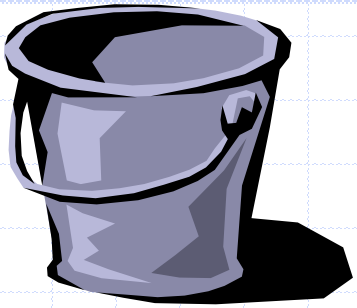
# Bucket-Sort and Radix-Sort



# Bucket-Sort

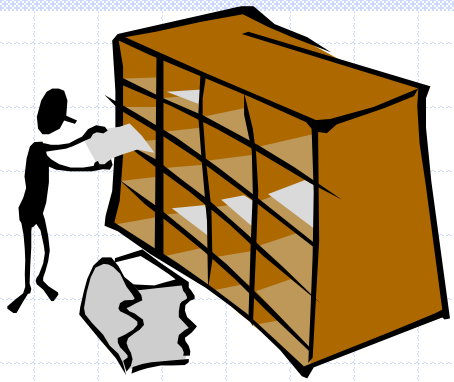


Problem: Sort a sequence $S$ which has $n$ items.

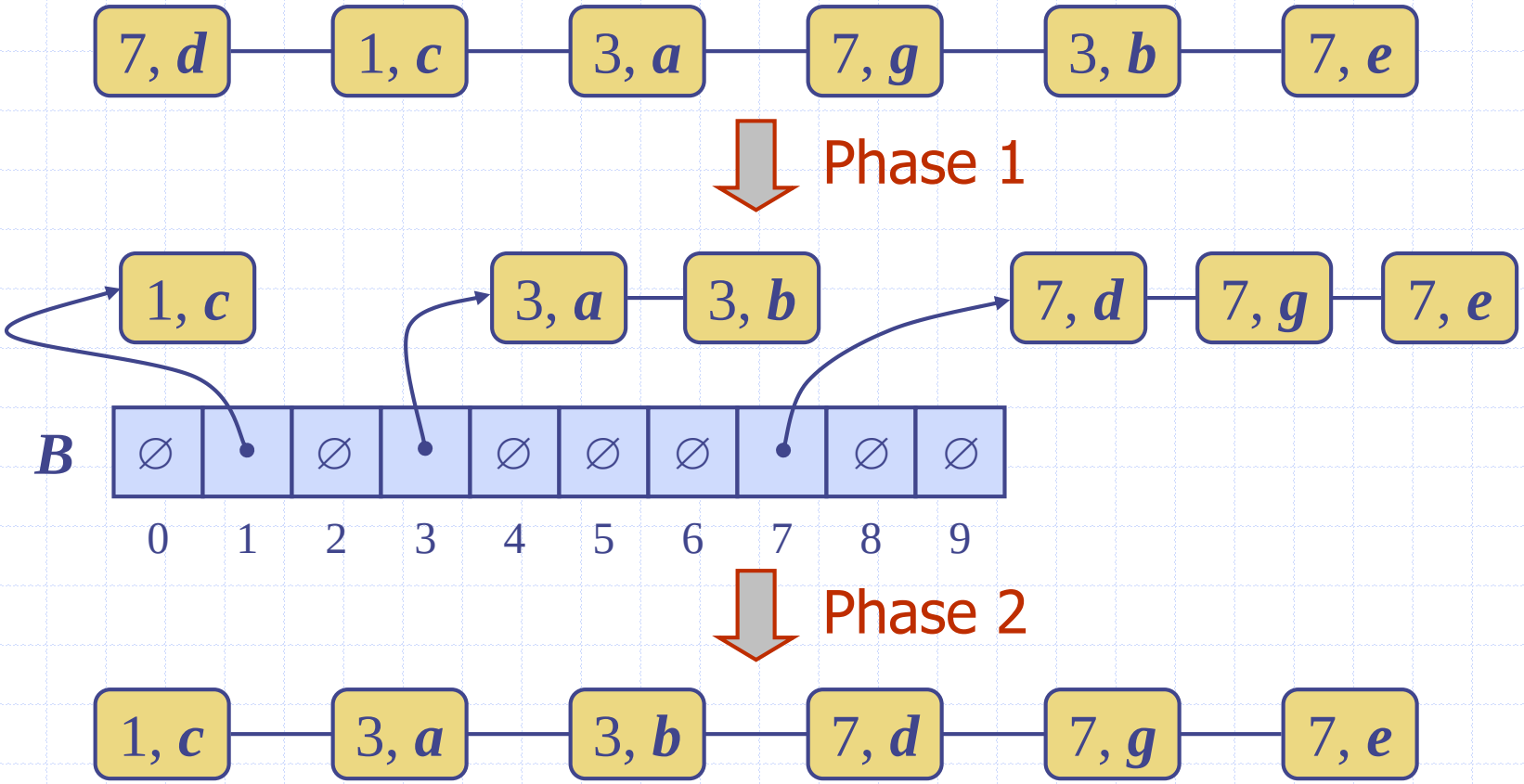
Condition: Each item has a key, and the items should be sorted based on their key values.

Range: The range of the key values is $[0, N-1]$

# Example



- 6 items with key range $[0, 9]$



# Properties and Complexity

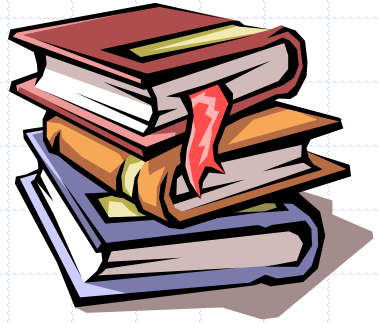
## Stable:

The relative order of any two items with the same key is preserved after the execution of the algorithm

## Complexity:

If there are n items and the range of keys is [0, N] then the complexity of bucket sort is O(n + N).

# Lexicographic Order

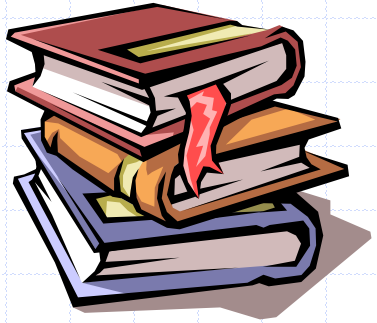


- A $d$-tuple is a sequence of $d$ keys $(k_1, k_2, \ldots, k_d)$, where key $k_i$ is said to be the $i$-th dimension of the tuple
- Example:
  - The Cartesian coordinates of a point in space are a 3-tuple
- The lexicographic order of two $d$-tuples is defined as follows

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d) \text{ if:}$$

$$(x_1 < y_1) \text{ or}$$
$$(x_1 = y_1 \text{ and } x_2 < y_2) \text{ or}$$
$$(x_1 = y_1 \text{ and } x_2 = y_2 \text{ and } x_3 < y_3) \text{ or } \ldots.$$

# Lexicographic Order



- A $d$-tuple is a sequence of $d$ keys $(k_1, k_2, \ldots, k_d)$, where key $k_i$ is said to be the $i$-th dimension of the tuple
- Example:
  - The Cartesian coordinates of a point in space are a 3-tuple
- The lexicographic order of two $d$-tuples is recursively defined as follows

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d)$$

$$\Leftrightarrow$$

$$x_1 < y_1 \vee x_1 = y_1 \wedge (x_2, \ldots, x_d) < (y_2, \ldots, y_d)$$

# Radix-Sort

- Radix-sort sorts a sequence of $d$-tuples in lexicographic order by executing $d$ times algorithm ***Bucket-sort***, one per dimension
- Radix-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of ***Bucket-Sort***

**Algorithm** *Radix-sort*($S$)

**Input** sequence $S$ of $d$-tuples
**Output** sequence $S$ sorted in lexicographic order

**for** $i \leftarrow 1$ **upto** d
*Bucket-sort*($S$, $C_i$)

Example:

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)

# Radix-Sort

- Radix-sort sorts a sequence of $d$-tuples in lexicographic order by executing $d$ times algorithm ***Bucket-sort***, one per dimension
- Radix-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of ***Bucket-Sort***

**Algorithm** ***Radix-sort***$(S)$

**Input** sequence $S$ of $d$-tuples
**Output** sequence $S$ sorted in lexicographic order

**for** $i \leftarrow 1$ **upto** d
***Bucket-sort***$(S, C_i)$

Example:

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)

This will result in wrong answer!!

# Radix-Sort (other way around)

- Radix-sort sorts a sequence of $d$-tuples in lexicographic order by executing $d$ times algorithm ***Bucket-sort***, one per dimension
- Radix-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of ***Bucket-Sort***

**Algorithm** *lexicographicSort*($S$)
  **Input** sequence $S$ of $d$-tuples
  **Output** sequence $S$ sorted in lexicographic order

  **for** $i \leftarrow d$ **downto** 1
    *stableSort*($S$, $C_i$)

Example:

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)

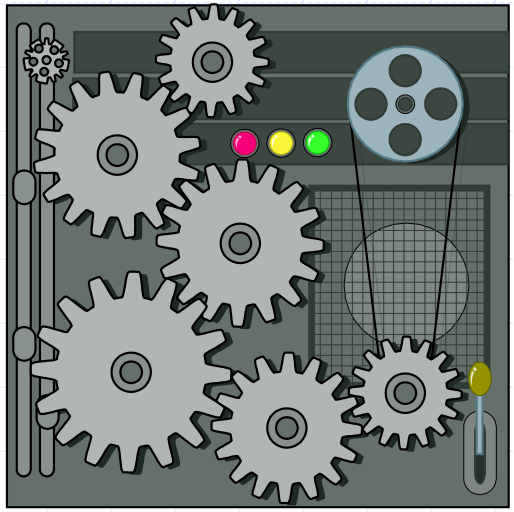
(2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)

(2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)

(2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)

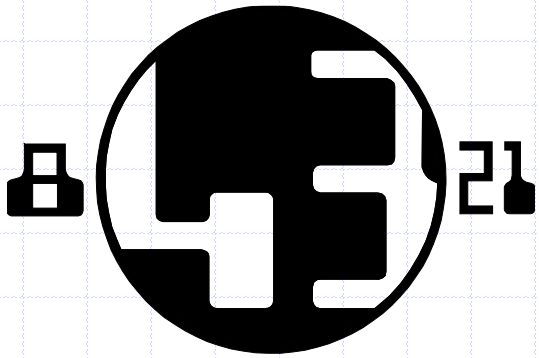
This is the correct way!

# Complexity



- Radix-sort runs in time $O(d(n + N))$

# Radix-Sort for Binary Numbers



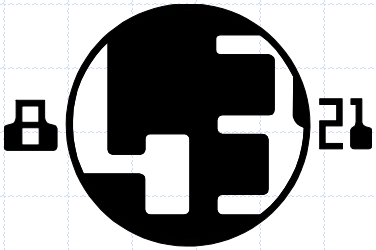

- Consider a sequence of $n$ $b$-bit integers

$$x = x_{b-1} \cdots x_1 x_0$$

- We represent each element as a $b$-tuple of integers in the range $[0, 1]$ and apply radix-sort with $N = 2$
- This application of the radix-sort algorithm runs in $O(bn)$ time
- For example, we can sort a sequence of 32-bit integers in linear time

**Algorithm** *binaryRadixSort*($S$)
  **Input** sequence $S$ of $b$-bit integers
  **Output** sequence $S$ sorted
  replace each element $x$ of $S$ with the item $(0, x)$
  **for** $i \leftarrow 0$ **to** $b - 1$
    replace the key $k$ of each item $(k, x)$ of $S$ with bit $x_i$ of $x$
    *bucketSort*($S$, 2)

# Example




- Sorting a sequence of 4-bit integers