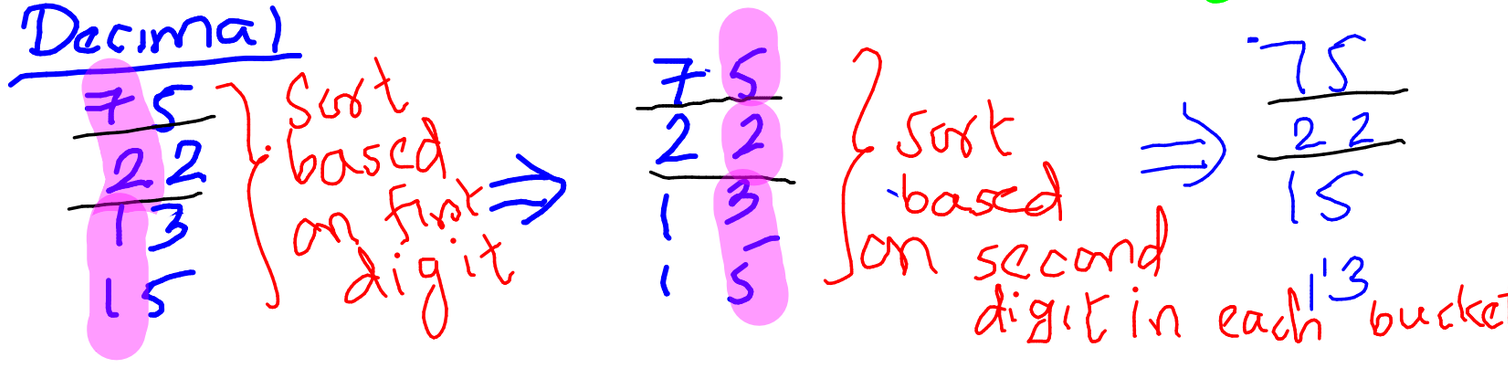
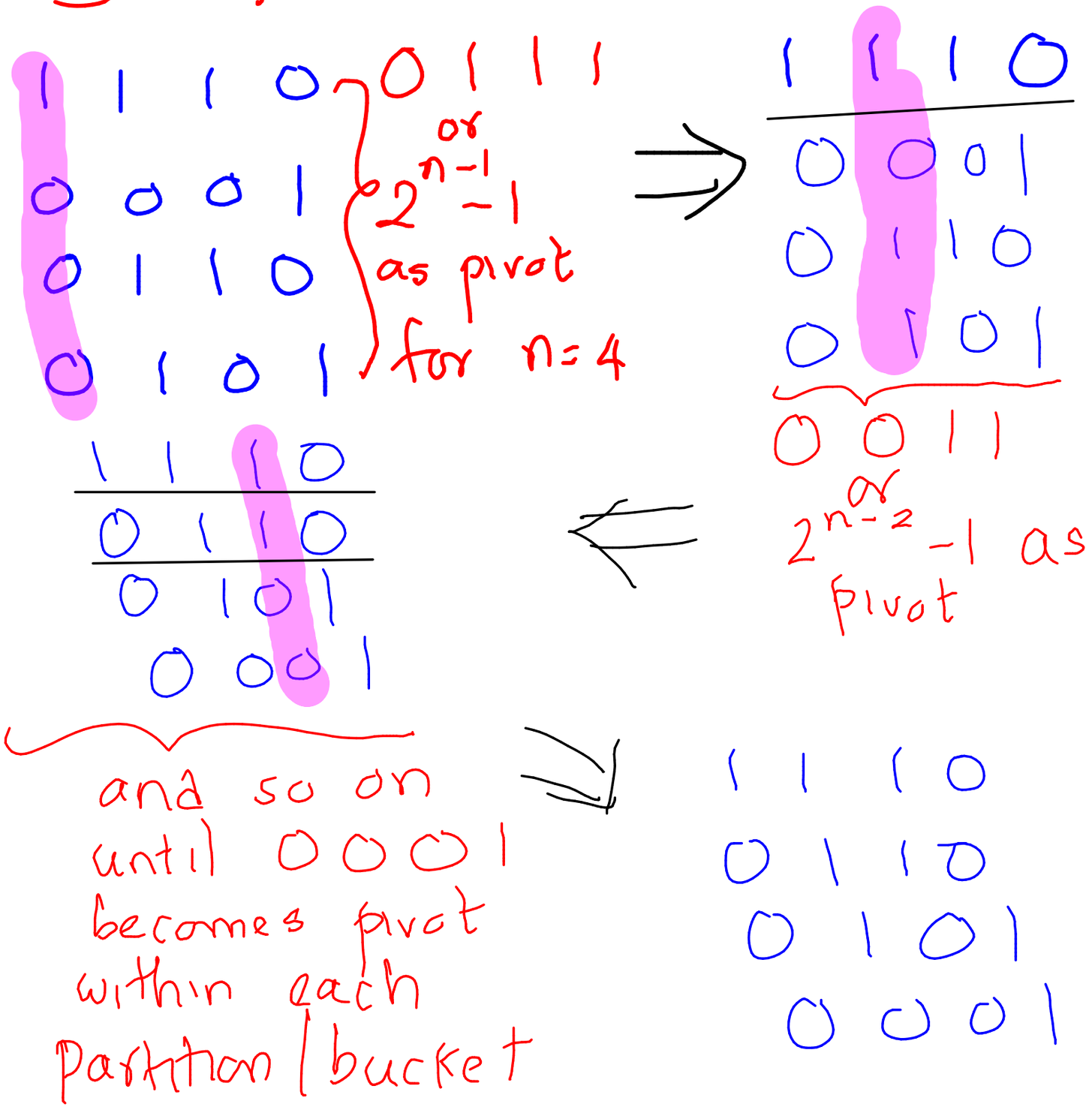
So far we considered 2 divide & conquer based strategies for Sorting

1. Divide at a fixed (centre) pt of array
2. Divide based on a (randomly chosen) pivot element
3. Divide based on bucketing → BUCKET SORT

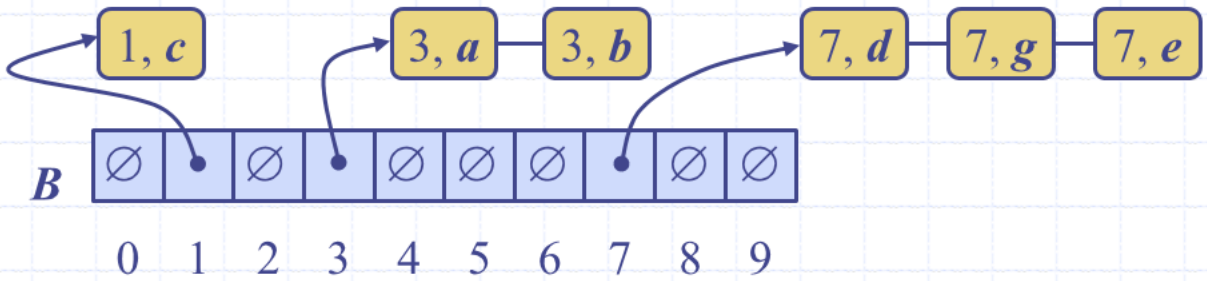


④ Divide based on fixed number as pivot (quicksort style) → RADIX SORT

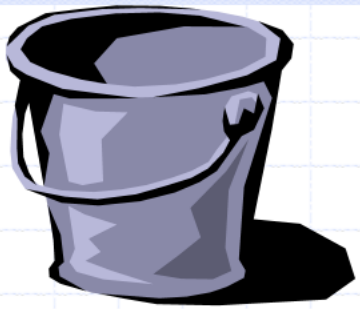
Binary representation



Bucket-Sort and Radix-Sort



Bucket-Sort (§ 10.4.1)



- Let be S be a sequence of n (key, element) entries with keys in the range $[0, N-1]$
- Bucket-sort uses the keys as indices into an auxiliary array B of sequences (buckets)
  - Phase 1: Empty sequence S by moving each entry (k, o) into its bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N-1$, move the entries of bucket $B[i]$ to the end of sequence S
- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n + N)$ time

  Bucket-sort takes $O(n + N)$ time

Algorithm *bucketSort*(S, N)

Input sequence S of (key, element) items with keys in the range $[0, N-1]$

Output sequence S sorted by increasing keys

$B \leftarrow$ array of N empty sequences

while $\neg S.isEmpty()$

$f \leftarrow S.first()$

$(k, o) \leftarrow S.remove(f)$

$B[k].insertLast((k, o))$

for $i \leftarrow 0$ **to** $N-1$

while $\neg B[i].isEmpty()$

$f \leftarrow B[i].first()$

$(k, o) \leftarrow B[i].remove(f)$

$S.insertLast((k, o))$

Eg: $S = [01 \quad 05 \quad 10 \quad 15]$ (assume only keys k are shown)

$N = 16$ $B[0, 1, 2 \ldots 15] = [\_, 01, \_\_ 05, \_\_]$

S = [01 05 10 15]

Phase 1: Empty S into B

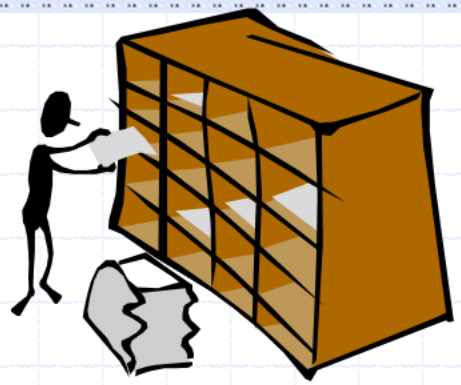
S = [ ]

B = [_, 01, _, _, _, 05, _, _, _, _, 10, _, _, _, _, 15]

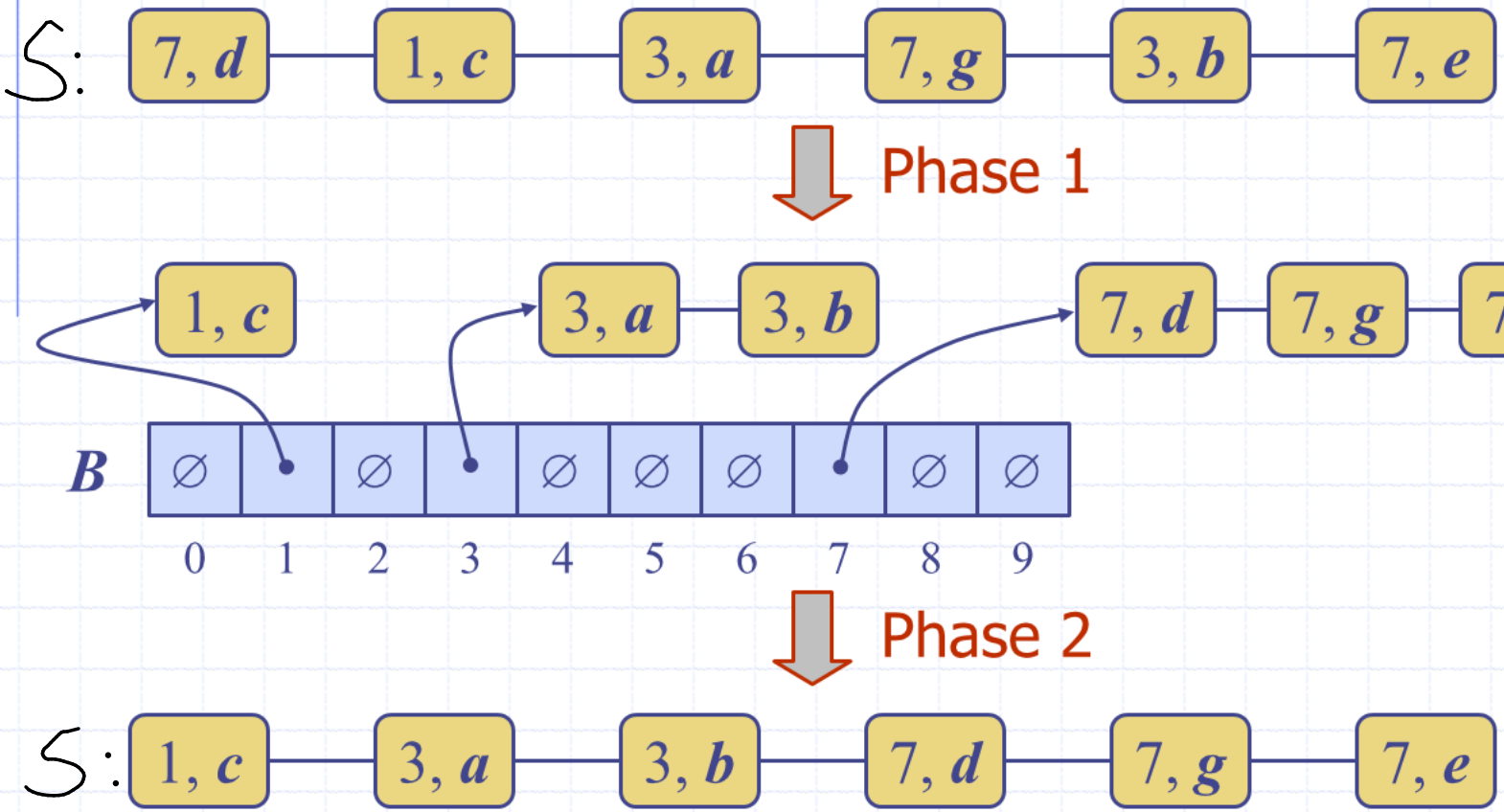
Phase 2: Scan B and append sequentially to S

S = [01, 05, 10, 15]

Example



- Key range [0, 9]



Properties and Extensions

- Key-type Property
  - The keys are used as indices into an array and cannot be arbitrary objects
  - No external comparator
- **Stable** Sort Property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm

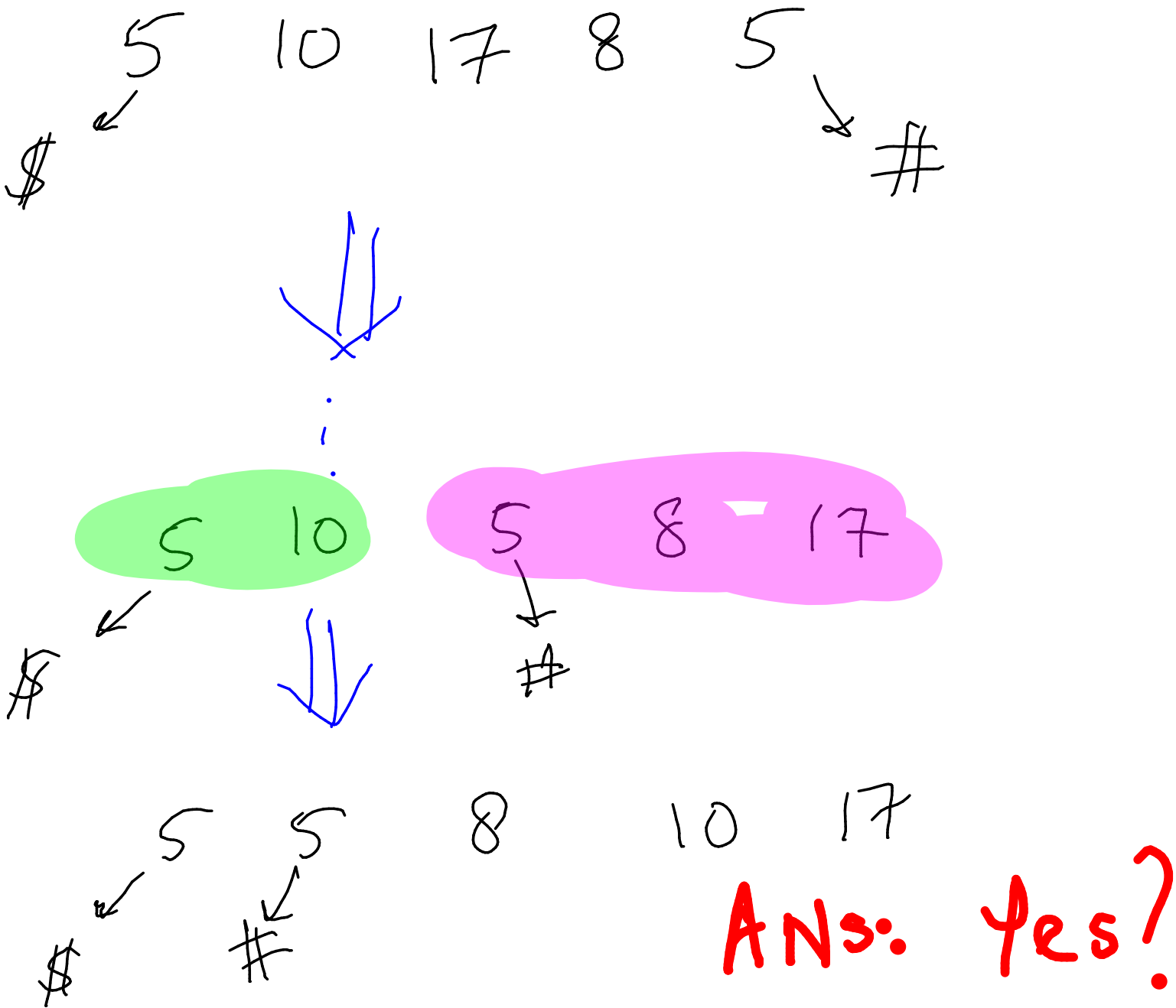
Extensions

- Integer keys in the range $[a, b]$
  - Put entry (k, o) into bucket $B[k - a]$
- String keys from a set D of possible strings, where D has constant size (e.g., names of the 50 U.S. states)
  - Sort D and compute the rank $r(k)$ of each string k of D in the sorted sequence
  - Put entry (k, o) into bucket $B[r(k)]$

1-1 monotone hash fn

Bucket sort is stable

Q: Is merge sort stable?



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

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

$$\Leftrightarrow$$

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

$x_1 = y_1$

I.e., the tuples are compared by the first dimension, then by the second dimension, etc.

Lexicographic-Sort

- Let C_i be the comparator that compares two tuples by their i-th dimension
- Let *stableSort*(S, C) be a stable sorting algorithm that uses comparator C
- Lexicographic-sort sorts a sequence of d-tuples in lexicographic order by executing d times algorithm *stableSort*, one per dimension
- Lexicographic-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of *stableSort*

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)