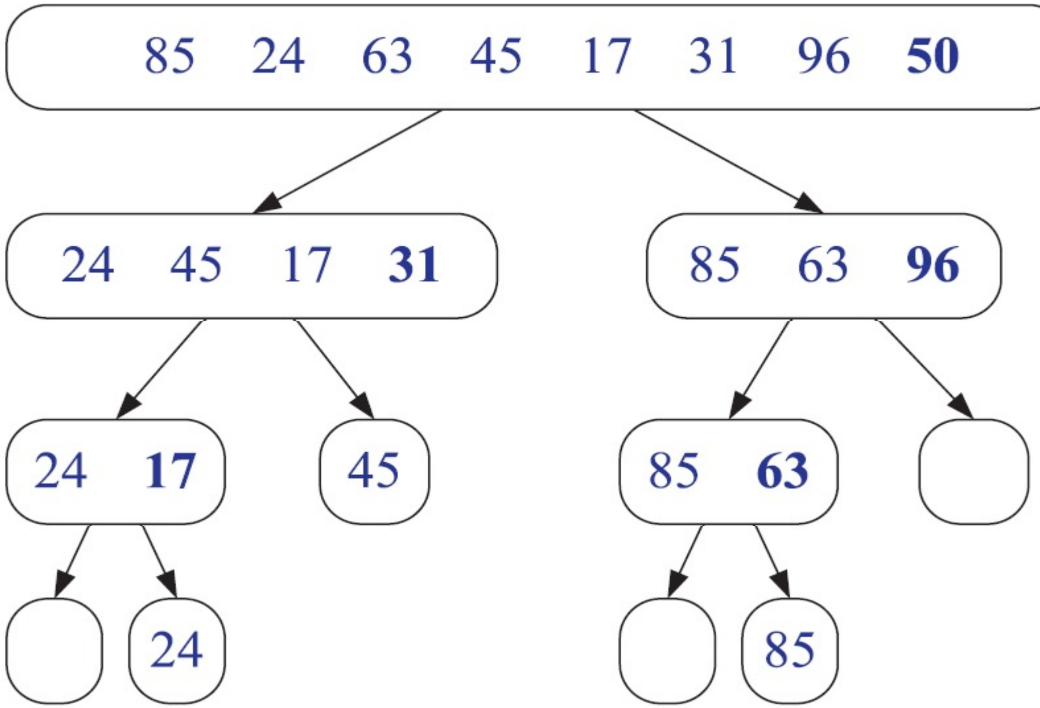
53

# IN-PLACE IMPLEMENTATION OF QUICK-SORT

# IN-PLACE IMPLEMENTATION

- Recall that an algorithm is **in-place** if it uses no additional memory space or only a small memory space in addition to the space needed for the objects being sorted themselves
- For quick-sort, the **in-place** implementation avoids copying elements into temporary memory spaces



# IN-PLACE IMPLEMENTATION

- Recall that an algorithm is **in-place** if it uses no additional memory space or only a small memory space in addition to the space needed for the objects being sorted themselves

- For quick-sort, the **in-place** implementation avoids copying elements into temporary memory spaces

- Instead, it moves objects around within the array itself…

$QuickSort(\ array\ A\ )$
- If $|A| \leq 1$, return $A$;
- $p = ChoosePivot(A)$
- $\boxed{Partition(\ A, p\ )}$
- $QuickSort($1st part$)$
- $QuickSort($2nd part$)$

$$\underbrace{L(< p)}_{1^{st}\text{ part}}\ \ p\ \ \underbrace{E\&G(\geq p)}_{2^{nd}\text{ part}}\quad A$$

# IN-PLACE IMPLEMENTATION

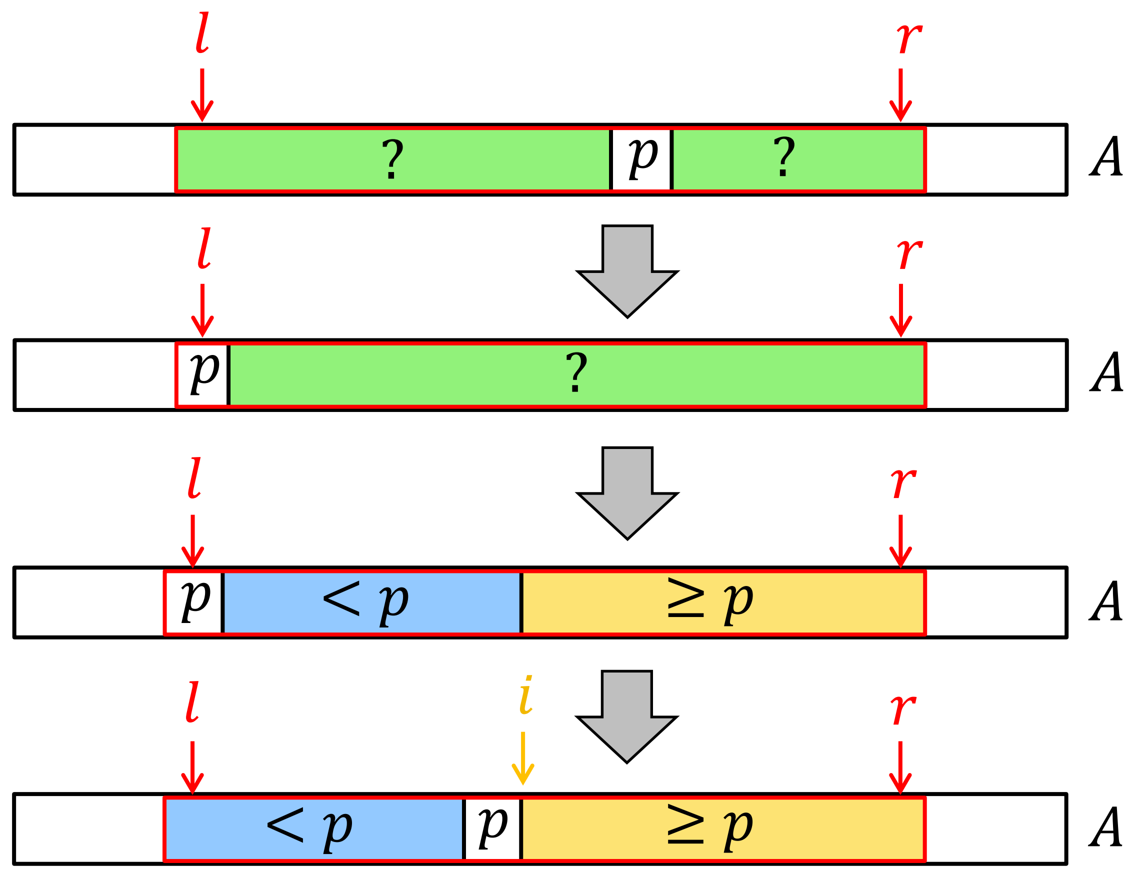
- Let $l$ be the left-most index in the part of A that is being sorted
- Let $r$ be the right-most index in the part of A that is being sorted

$Partition($ A, $p, l, r$ $)$

swap $A[pivotIndex]$ and $A[l]$

Split the elements between ($l$+1) and $r$ into a part that is $< p$ and a part that is $\geq p$

Swap $A[pivotIndex]$ and $A[i-1]$




Let's see how the pink box works…

# IN-PLACE IMPLEMENTATION

$Partition(\,A, p, l, r\,)$

swap $A[pivotIndex]$ and $A[l]$

$i \leftarrow (l + 1)$

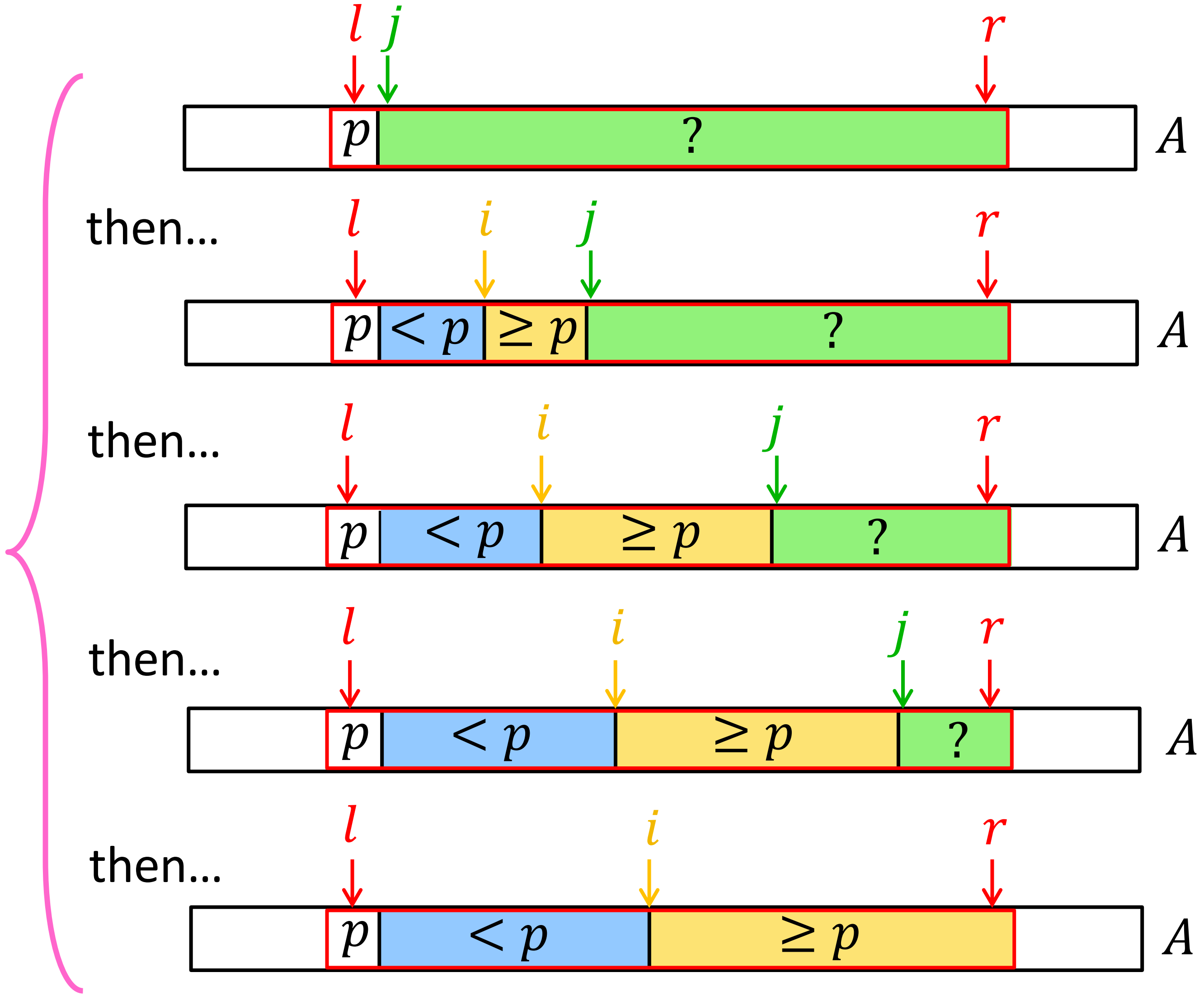
for $j \leftarrow (l + 1)$ to $r$ do

If $A[j] < p$ and $A[i] > p$

Swap $A[j]$ and $A[i]$

$i$++

Swap $A[pivotIndex]$ and $A[i - 1]$



# SORTING IN LINEAR TIME

# LINEAR-TIME SORTING

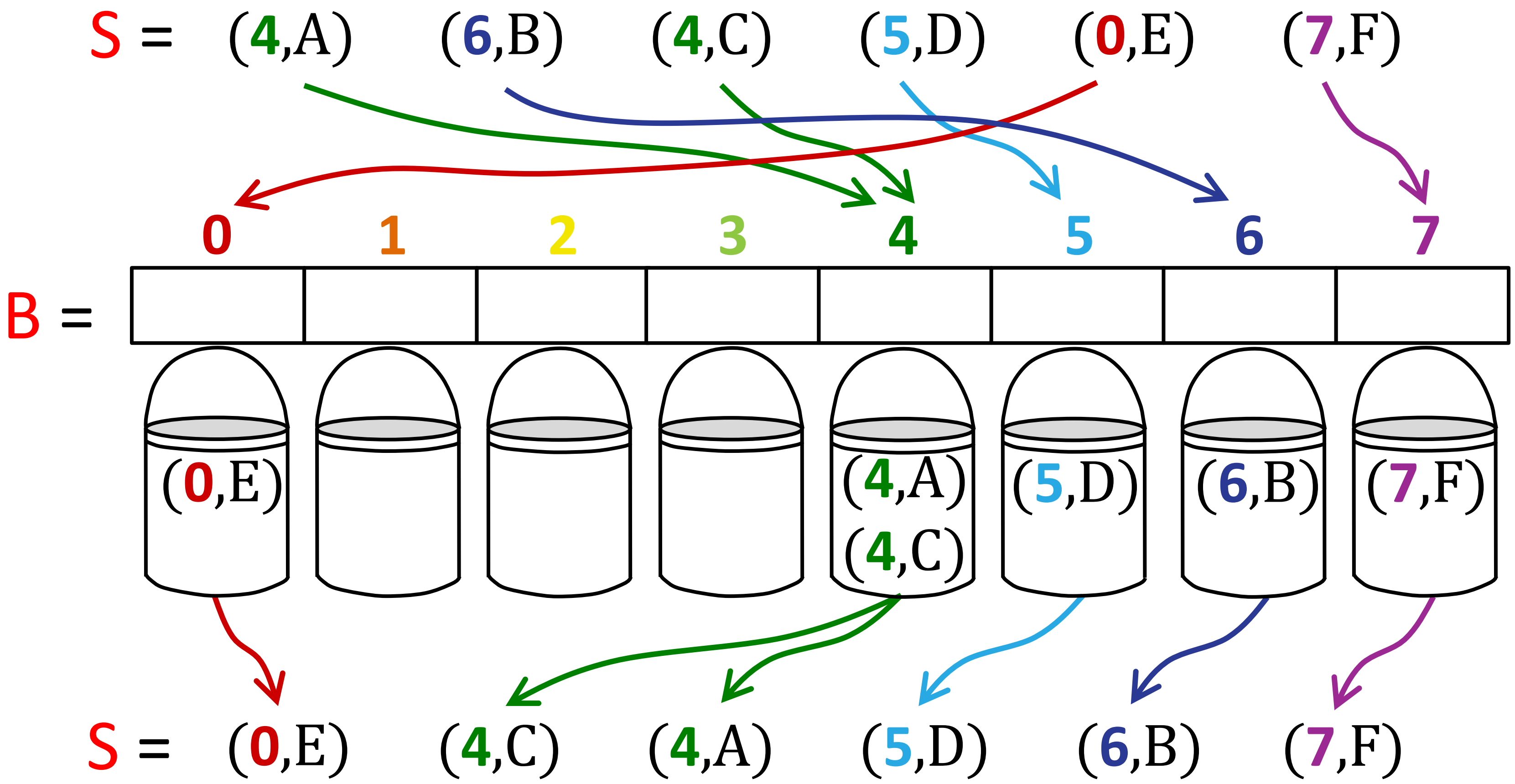
- All the sorting algorithms we've seen so far are "**comparison-based**" algorithms, i.e., they are based on comparing elements to each other.
- It is proven that no comparison-based algorithm can sort $n$ elements in less than $O(n \log n)$ time!
- Interestingly, there are some sorting algorithms that are:
  - Not comparison-based!
  - Run in $O(n)$ time!

but they require special assumptions about the input sequence to be sorted!

Let's look at 2 examples of such algorithms …

# BUCKET-SORT

- Given a sequence S containing n entries whose keys are **integers** in the range [0,N−1], **bucket-sort** is a sorting algorithm that sorts S using a **bucket array**, B:



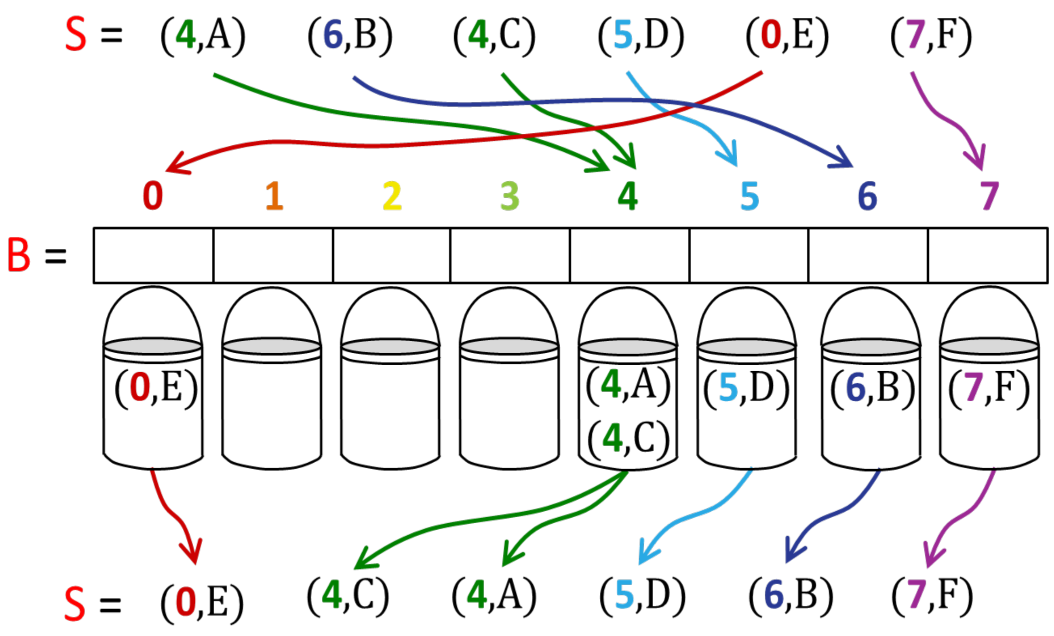
# BUCKET-SORT

**Algorithm** bucketSort($S$):

**Input:** Sequence S of entries with integer keys in the range [0,N −1]

**Output:** $S$ sorted in nondecreasing order of the keys

```
for each entry e in S do
    k ← e.key()
    remove e from S and insert it at B[k]
for i ← 0 to N −1 do
    for each entry e in sequence B[i] do
        remove e from B[i] and insert it at the end of S
```



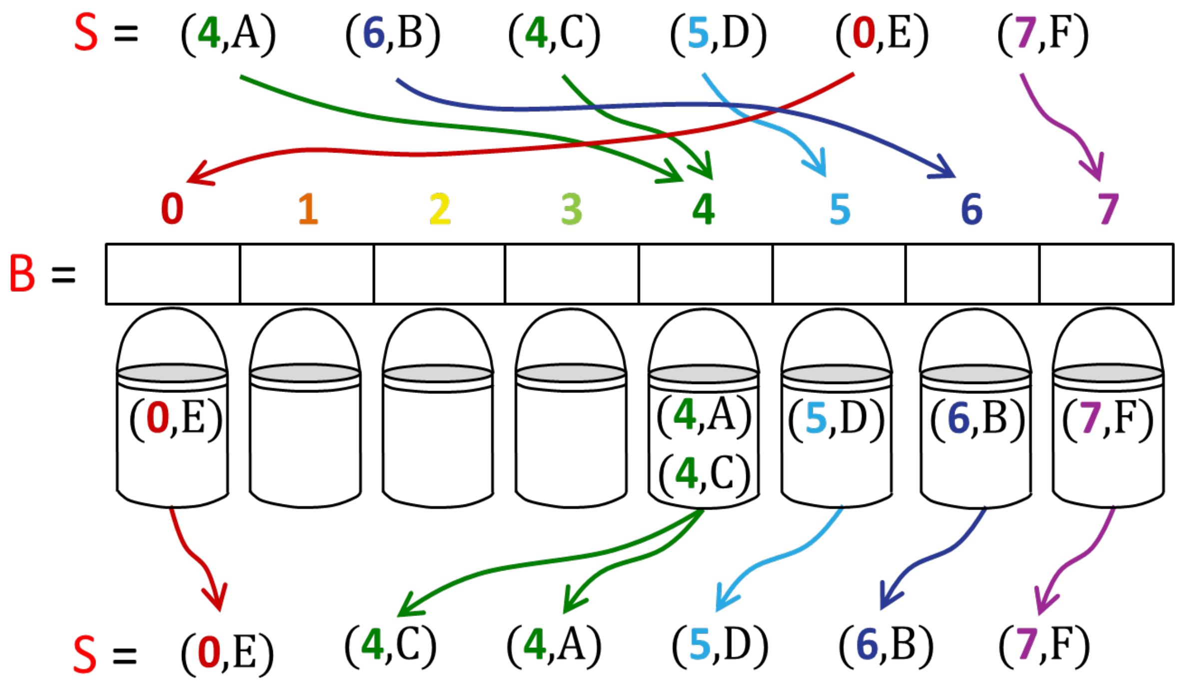

- What is the complexity? O(n+N)
- Hence, if N itself is O(n), then bucket-sort runs in linear time!
- However, the performance deteriorates as N grows compared to n

# STABLE SORTING

- Given a sequence $S = ((k_0, x_0), \ldots, (k_{n-1}, x_{n-1}))$, a sorting algorithm is said to be "**stable**" if, for any two entries $(k_i, x_i)$ and $(k_j, x_j)$ of $S$, such that $k_i = k_j$:
  - If $(k_i, x_i)$ precedes $(k_j, x_j)$ in $S$ **before** sorting, then $(k_i, x_i)$ also precedes $(k_j, x_j)$ **after** sorting.

- Our initial description of bucket-sort does not guarantee stability; notice how (4,A) precedes (4,C) before the sorting, but ends up after (4,C) after the sorting

- This can easily be fixed by always removing the first element in the bucket!



# RADIX-SORT

- Consider a sequence

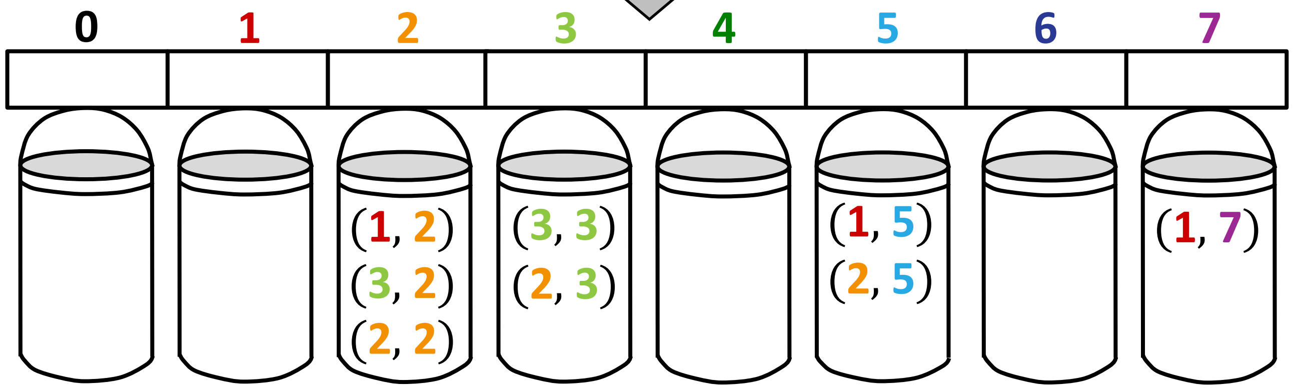
  $S = \big((k_0, l_0), x_0\big), \big((k_1, l_1), x_1\big), \ldots$

  where every key consists of **two integer parts** that are in [0,N−1]

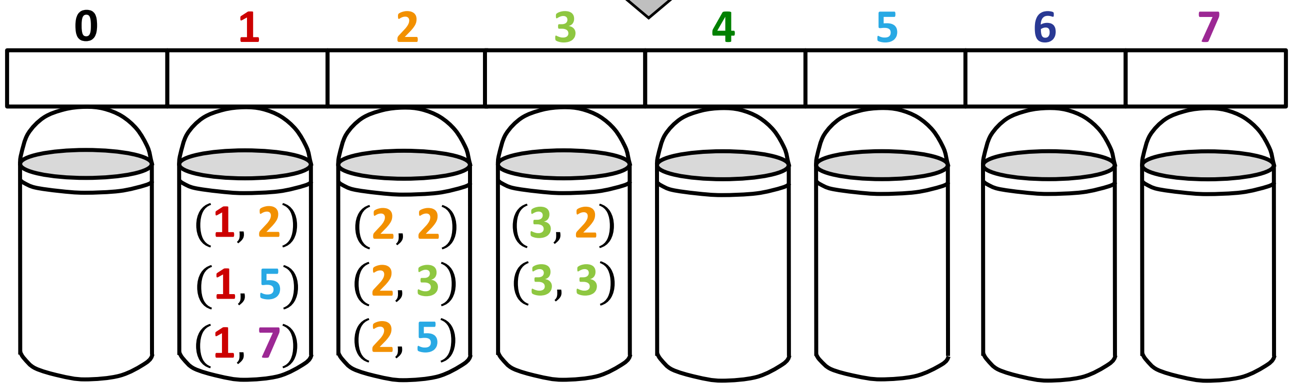
- Radix-sort is an algorithm that sorts a sequence such as S **lexicographically** by applying stable bucket-sort on S twice:
  - ➢ once using the $l_i$ part, then
  - ➢ another using the $k_i$ part

- Given keys consisting of d parts, radix-sort runs in $O(d(n+N))$

$S = (3, 3),\ (1, 5),\ (2, 5),\ (1, 2),\ (2, 3),\ (1, 7),\ (3, 2),\ (2, 2)$




$S = (1, 2),\ (3, 2),\ (2, 2),\ (3, 3),\ (2, 3),\ (1, 5),\ (2, 5),\ (1, 7)$




$S = (1, 2),\ (1, 5),\ (1, 7),\ (2, 2),\ (2, 3),\ (2, 5),\ (3, 2),\ (3, 3)$

64

# COMPARING SORTING ALGORITHMS

# COMPARING SORTING ALGORITHMS

- **Insertion-sort:** If implemented well, the running time $O(n+m)$, where m is the number of **inversions** (i.e., pairs of elements out of order). Thus, it can be effective to sort sequences that are already **almost sorted**. But the worst case is $O(n^2)$!
- **Merge-sort:** Runs in $O(n \log n)$, but it's hard to implement it **in-place**. Thus, it needs more memory than what is needed to store the sequence itself! Also, there is an overhead due to repeatedly copying elements in memory, which slows it down slightly.
- **Quick-sort:** Its expected time is $O(n \log n)$, and in practice it is faster than merge-sort, making it an excellent choice as a general-purpose, **in-place** sorting algorithm. The only issue is the worst-case runtime, which is $O(n^2)$
- **Heap-sort:** Excellent; it runs in $O(n \log n)$, and can easily be made to execute **in-place**.
- **Bucket-sort:** If we have **integer keys** taken from [0,N−1] where N is small, then it can run in linear time if N itself is O(n)!

66

# FINAL EXAM LOGISTICS

# FINAL EXAM

- The final exam is on May 18th at 08:30am - 10:30am Abu Dhabi time.
- It is worth 30 points.
- The exam will be an **open book exam**, where you have access to your textbook, slides and notes only.
- No mobile phones, tablets or laptops are allowed.

# MATERIAL

- To prepare for the final, study:
  - All lecture/lab slides/notes
  - All lecture/lab examples/problems, quizzes, and assignments
  - All readings from the textbook

# EXAM FORMAT

- Similar to the midterm exam sitting, with the following types of questions:
  - Multiple choice questions
  - Short answer questions
  - Fix the code
  - Write a few lines of code

- GOOD LUCK!!!