

# Sorting II - Advanced

Unsorted Array

9	1	3	2	7	4
---	---	---	---	---	---



sorting algorithm

Sorted Array

1	2	3	4	7	9
---	---	---	---	---	---

# Learning Objectives

- Understand the basic principles and algorithms behind merge sort and quicksort.
- Compare and contrast the time complexity of merge sort, quick sort, bucket sort, and cyclic sort, including best-case, worst-case, and average-case scenarios.
- Identify the strengths and weaknesses of each sorting algorithm in terms of stability, adaptability to different data distributions, and ease of implementation.
- Explore potential optimizations and variations of merge sort, quick sort, bucket sort, and cyclic sort, such as parallelization, hybrid algorithms, and memory management techniques.

# Lecture Flow

- 1) Pre-requisites
- 2) Revision
- 3) Part I
  - Merge Sort
  - Bucket Sort
- 4) Part II
  - Quick Sort
  - Cyclic Sort
- 5) Practice Questions
- 6) Resources
- 7) Quote of the Day

# Pre-requisites

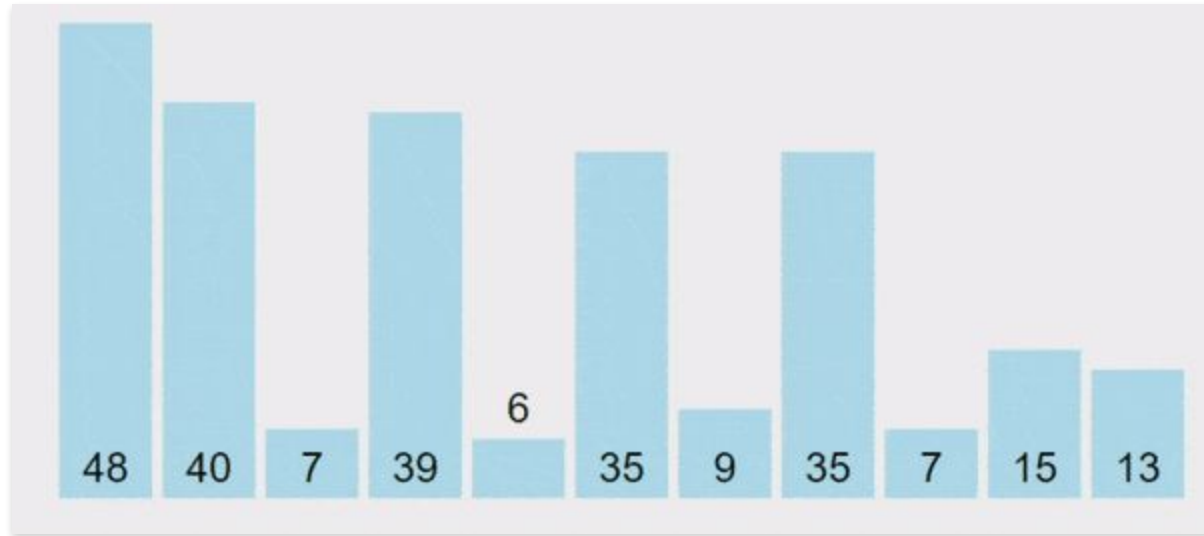
- Sorting - Basics
- Asymptotic Analysis
- Arrays
- Willingness to learn



# Revision



## Bubble Sort

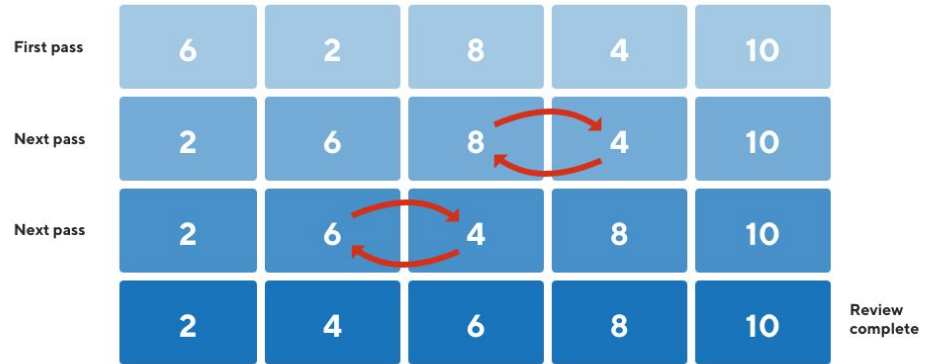


# Time & Space Complexity

Time complexity ? \_\_\_\_\_

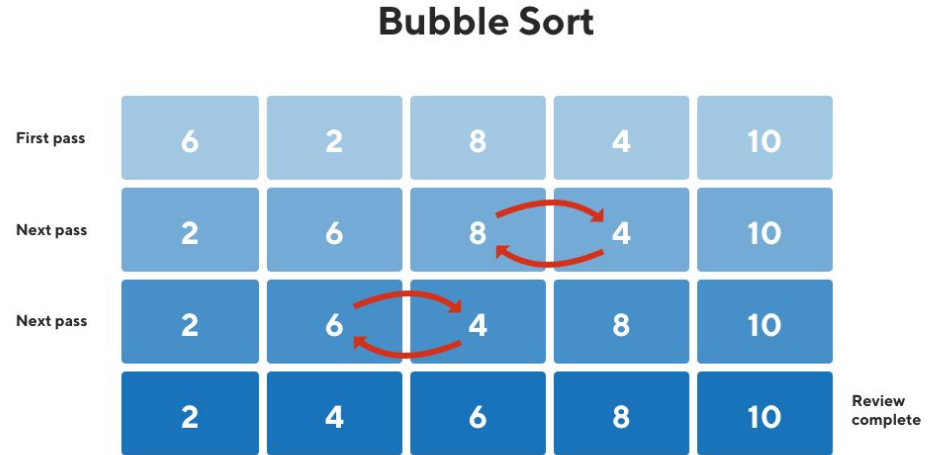
Space complexity ? \_\_\_\_\_

## Bubble Sort



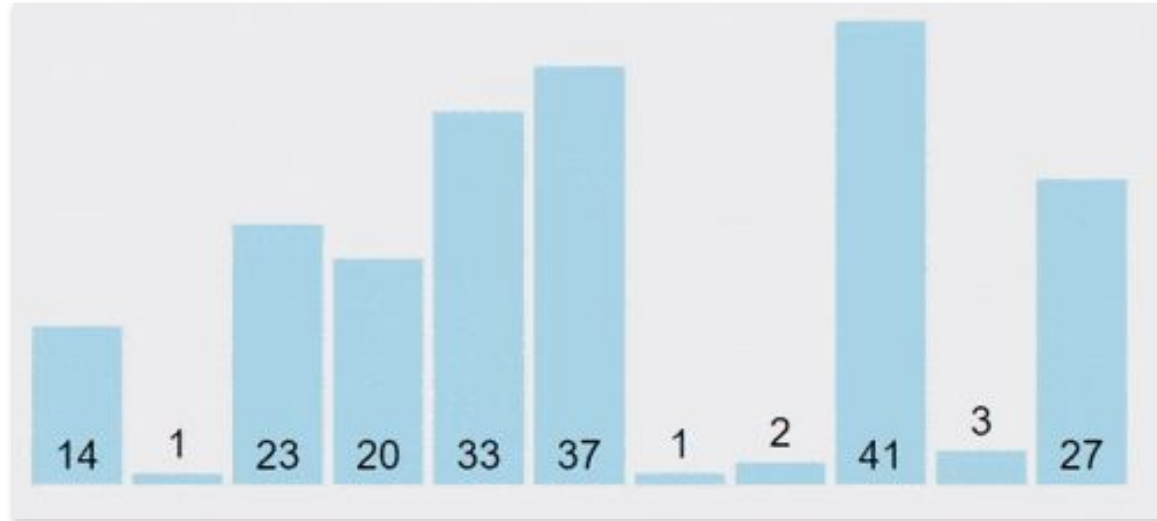
# Time & Space Complexity

Time complexity:  $O(n^2)$   
Space complexity:  $O(1)$





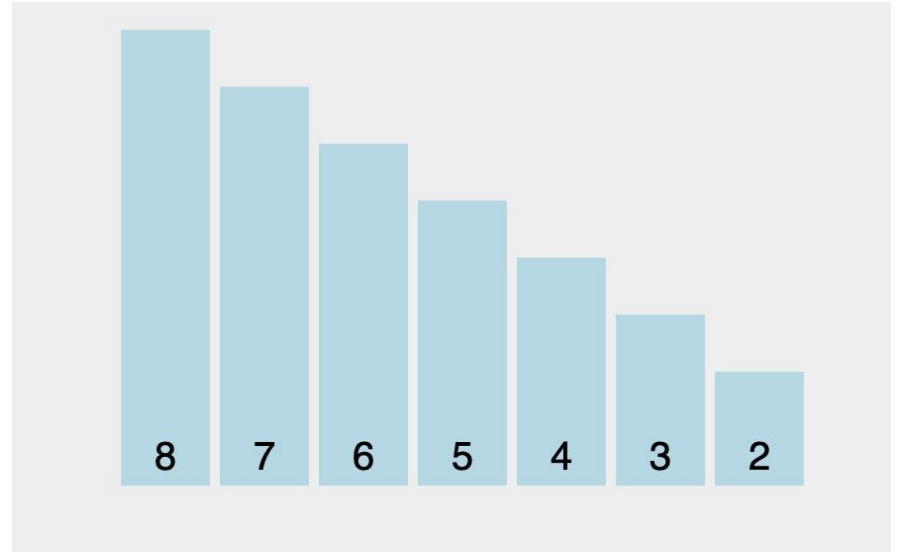
## Selection Sort



# Time & Space Complexity

Time complexity ? \_\_\_\_\_

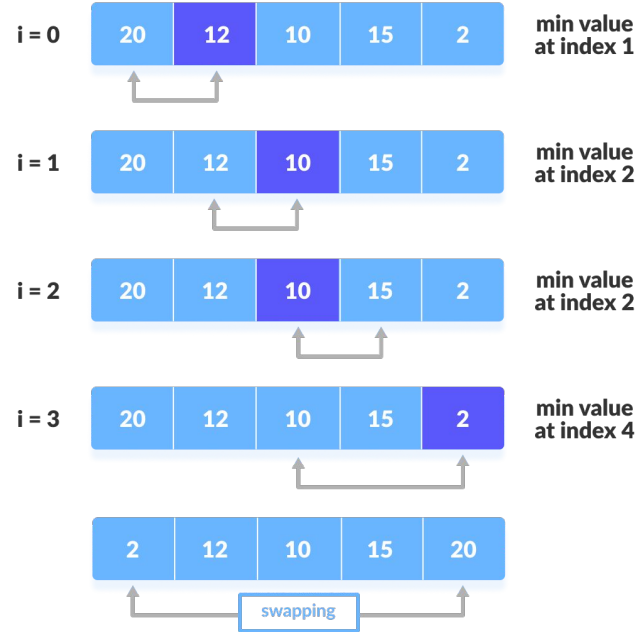
Space complexity ? \_\_\_\_\_



# Time & Space Complexity

Time complexity:  $O(n^2)$   
Space complexity:  $O(1)$

step = 0



# Insertion Sort



# Time & Space Complexity

Worst case ?

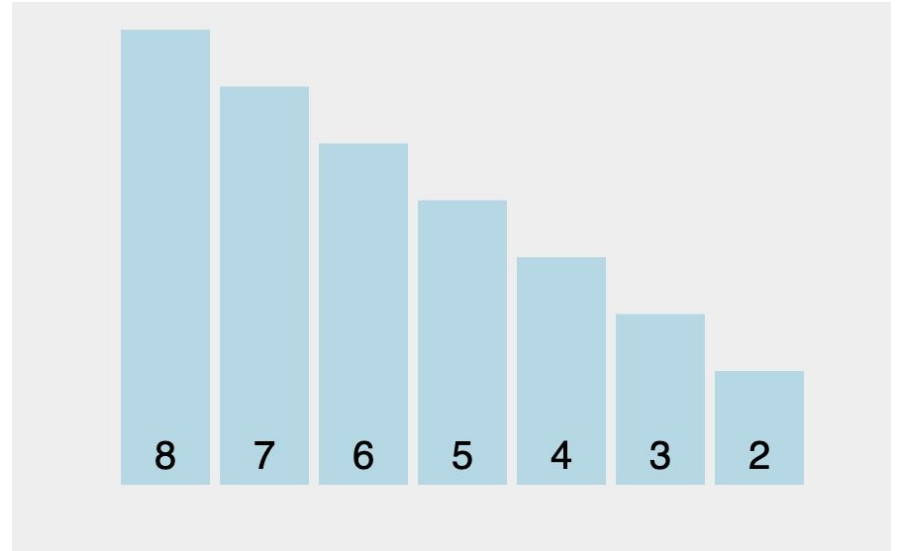
---

Best case ?

---

Average case ?

---



# Time & Space Complexity

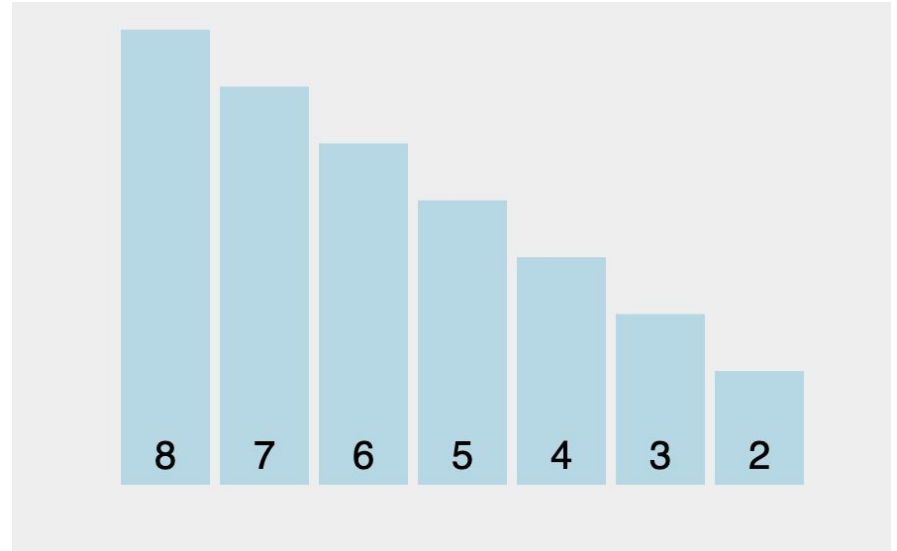
Time complexity:  **$O(n^2)$**

Space complexity:  **$O(1)$**

Worst case       **$O(n^2)$**

Best case       **$O(n)$**

Average case       **$O(n^2)$**



# Counting Sort



## Time complexity



Worst case ?

\_\_\_\_\_

Best case ?

\_\_\_\_\_

Average case ?

\_\_\_\_\_



## Time complexity

The time complexity of counting sort algorithm is  $O(n+k)$  where  $n$  is the number of elements in the array and  $k$  is the range of the elements.

Worst case	<u><math>O(n+k)</math></u>
Best case	<u><math>O(n+k)</math></u>
Average case	<u><math>O(n+k)</math></u>

**Note:** Counting sort is most efficient if the range of input values is not greater than the number of values to be sorted.

# Time to get efficient !



# Part I

# Merge Sort



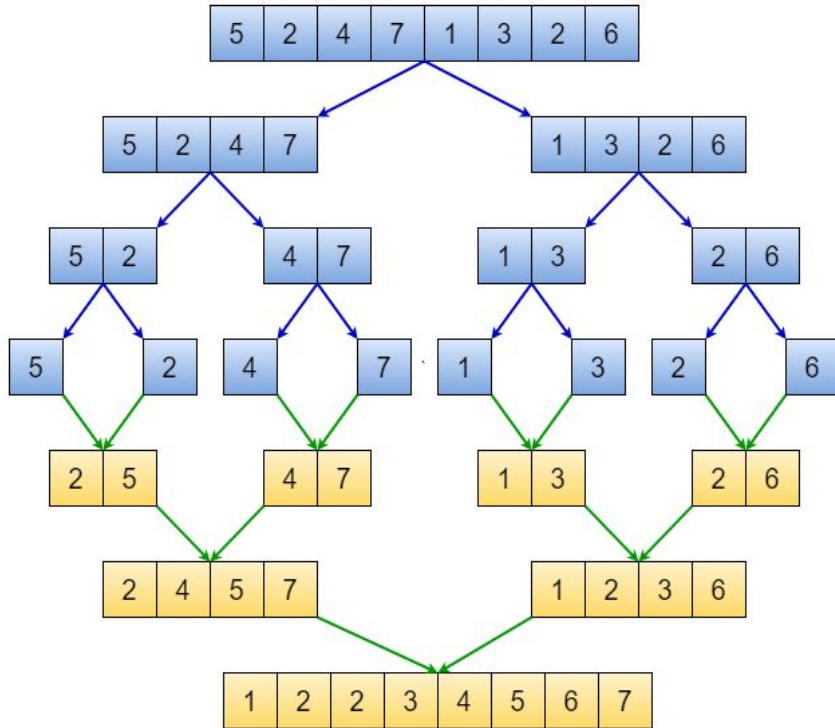
# Merge Sort



A sorting algorithm that works by **dividing** an array into smaller subarrays, **sorting each subarray**, and then merging the sorted subarrays **back together** to form the final sorted array.

visualization from: [VisuAlgo](#)

# Merge Sort



- Divide the array into two halves,
- Sort each half, and then
- Merge the sorted halves back together.

**Q:** Can you guess the next set of moves in the following dance ?



# Divide and Conquer

## Divide

- Divide the array into two

## Conquer

- Sort both halves with merge sort
- Base case?

## Combine

- Merge the two sorted halves



# Practice



Can you implement the function **merge** ?

**Implement Here**

Q: Is Merge Sort a **Stable** Sorting Algorithm ?



**Q:** What do you think is the time complexity for the aforementioned sorting Algorithm?



# Time & Space Complexity

Worst case

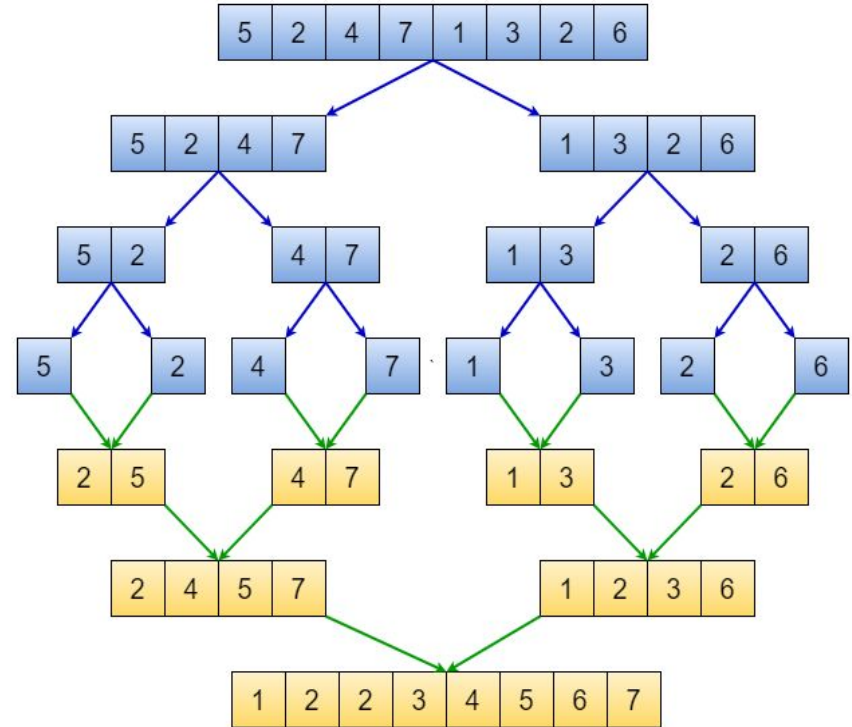
Best case

Average case

---

---

---



# Time & Space Complexity

Time complexity:  **$O(n \log n)$**

Space complexity:  **$O(n)$**

Worst case

**$O(n \log n)$**

Best case

**$O(n \log n)$**

Average case

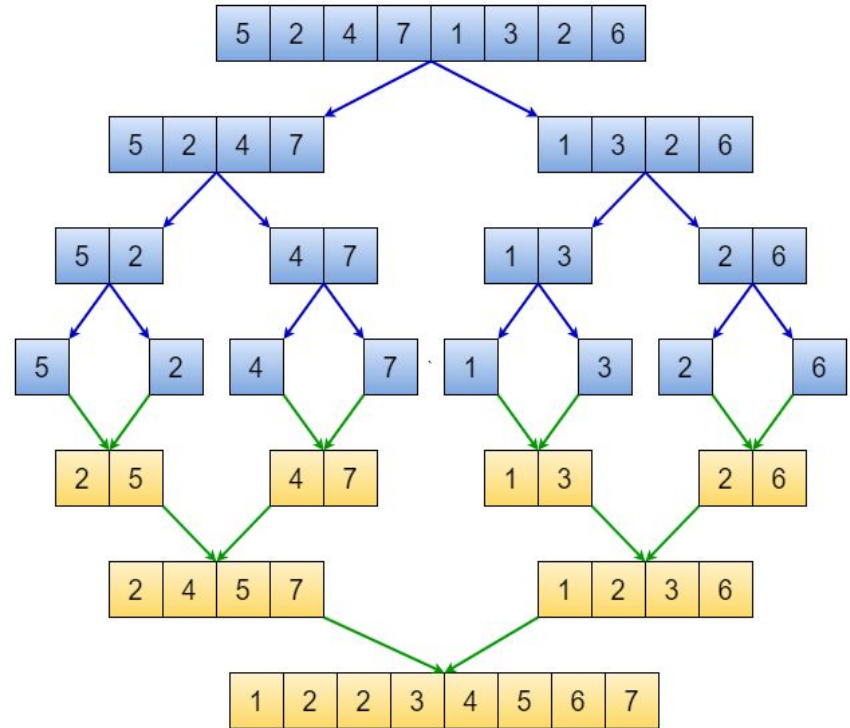
**$O(n \log n)$**

Stable

**YES**

In-place

**NO**



# Pair Programming

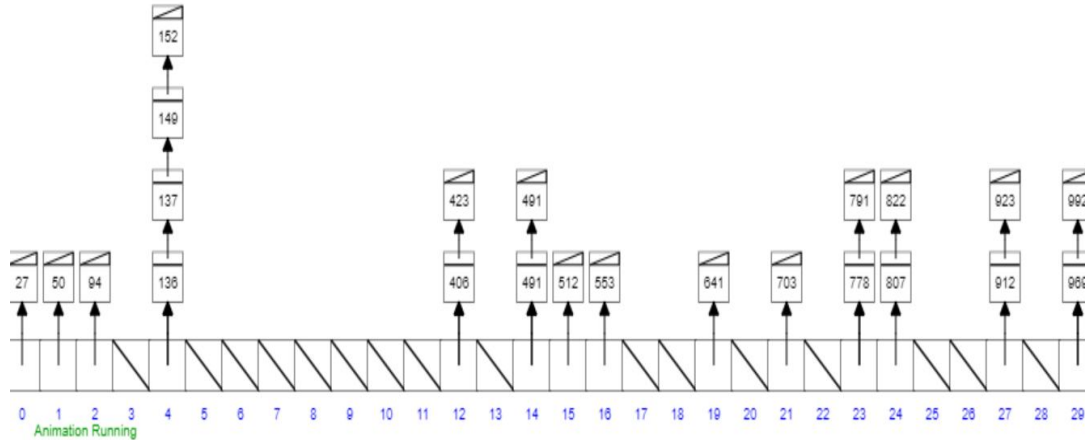
## Question 1

# Bucket Sort





# Bucket Sort



A sorting algorithm that works by **distributing** the elements of an array into a number of **buckets**, and then **sorting each bucket individually**. It is an efficient algorithm for sorting elements that are **evenly distributed across a range of values**.

# Bucket Sort

Here's how it works:

1. Determine the **range of values** in the array to be sorted.
2. **Divide** the range into a set of buckets.
3. For each element in the array, determine which bucket it **belongs** to and **insert** it into that bucket.
4. **Sort** each bucket **individually** using another sorting algorithm (usually insertion sort).
5. **Concatenate** the sorted buckets to obtain the final sorted array.

# Bucket Sort

## Problem:

Sort a large set of floating point numbers which are in **range** from **0.0** to **1.0** and are **uniformly** distributed across the range. How do we sort the numbers efficiently?

# Bucket Sort

## Approach:

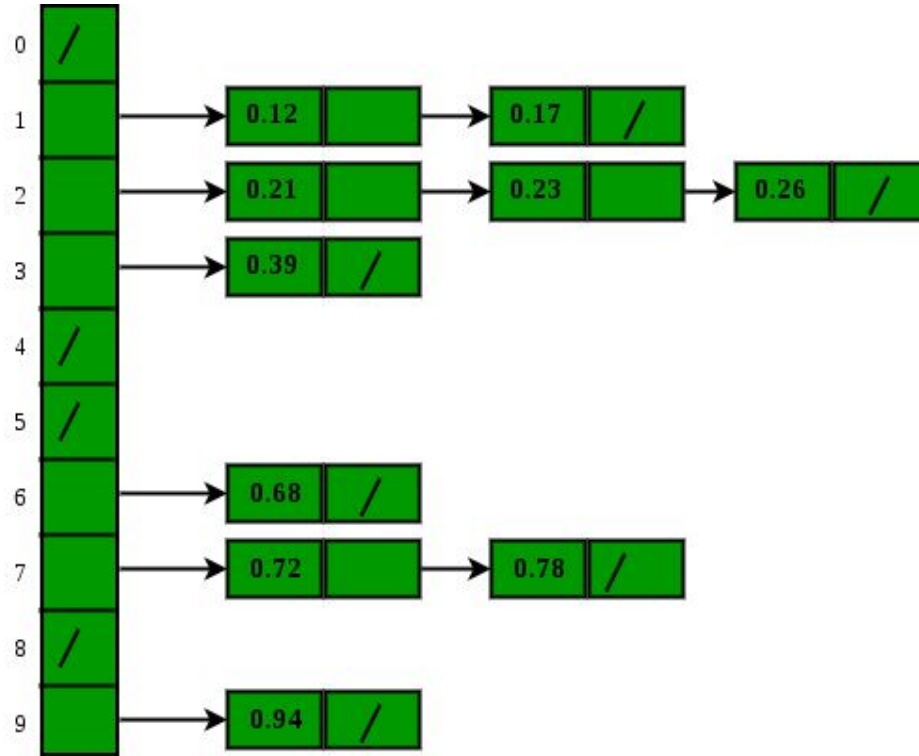
**bucket\_sort(arr[], n)**

- 1) Create n empty buckets (Or lists).
- 2) Do the following for every array element arr[i].
  - a) Insert arr[i] into bucket [n\*array[i]]
- 3) Sort individual buckets using insertion sort.
- 4) Concatenate all sorted buckets.

# Bucket Sort

0	0.78
1	0.17
2	0.39
3	0.26
4	0.72
5	0.94
6	0.21
7	0.12
8	0.23
9	0.68

Input Array



Buckets created

# Visualization Link



Can you implement the function **bucket\_sort** ?

Implement Here

# Time & Space Complexity

Worst case ?

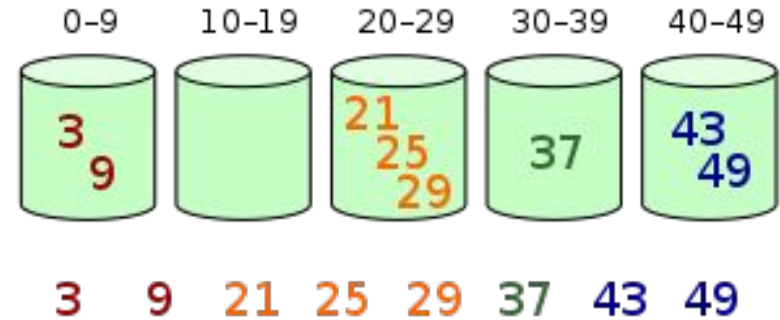
---

Best case ?

---

Average case ?

---





# Time & Space Complexity

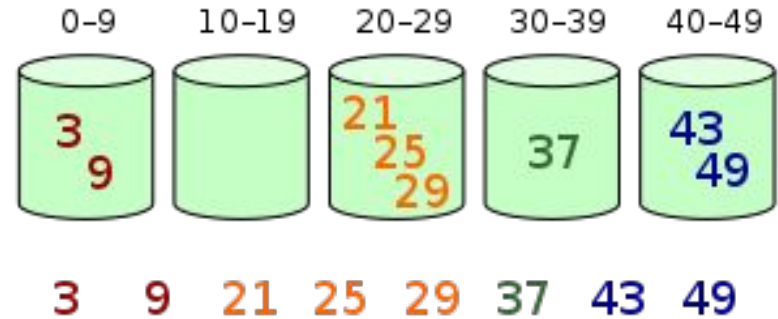
Time complexity:  $O(n^2)$

Space complexity:  $O(n + k)$

Worst case  $O(n^2)$

Best case  $O(n)$

Average case  $O(n + k)$



# Pair Programming

## Question 2

# Practice Problems

Sort List

Masha and Beautiful Tree

Count of Smaller Numbers After Self

Number of Pairs Satisfying Inequality

Create Sorted Array through Instructions

## Quote of the Day

"The first step in crafting a life you want is to get rid of everything you don't."

- **Joshua Becker**