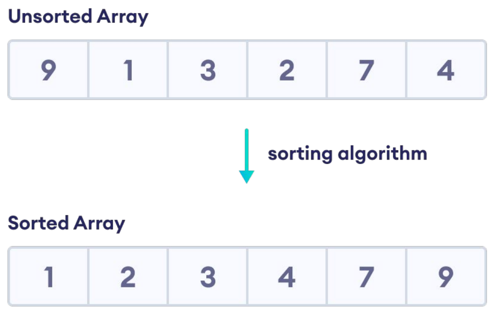
Sorting II - Advanced



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.

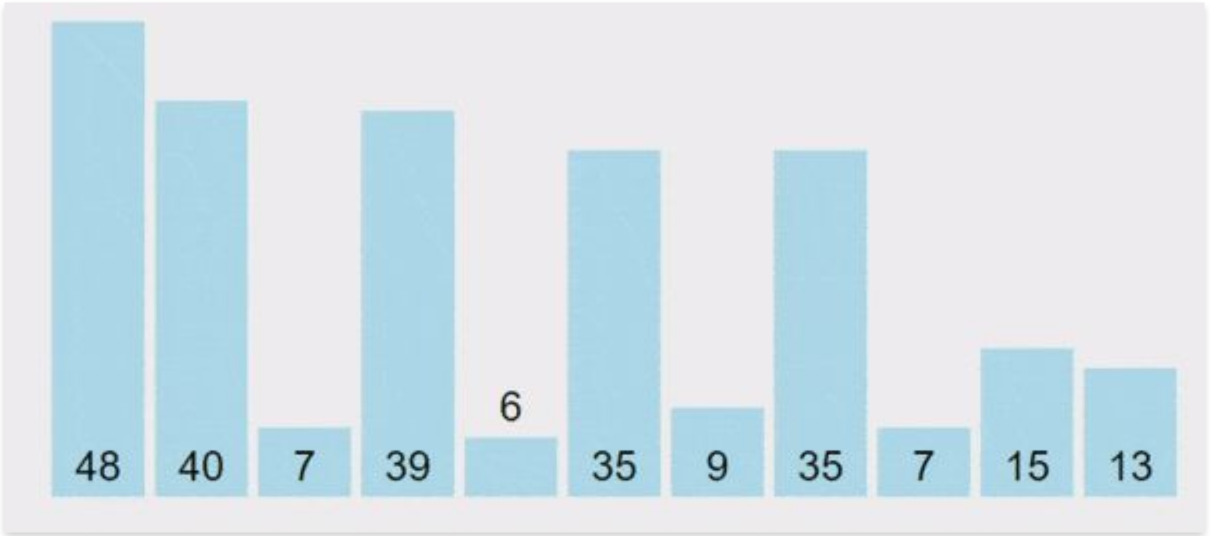
Lecture Flow

Pre-requisites

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

Revision

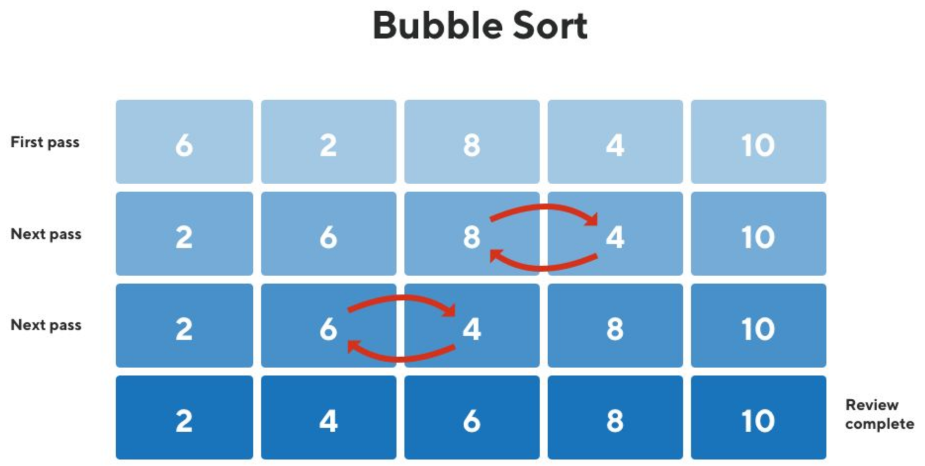
Bubble Sort



Time & Space Complexity

Time complexity ? ____________

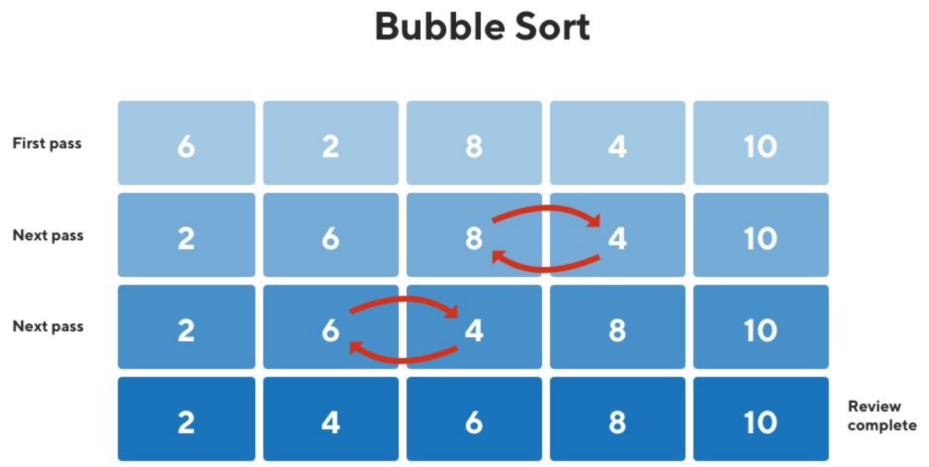
Space complexity ?____________



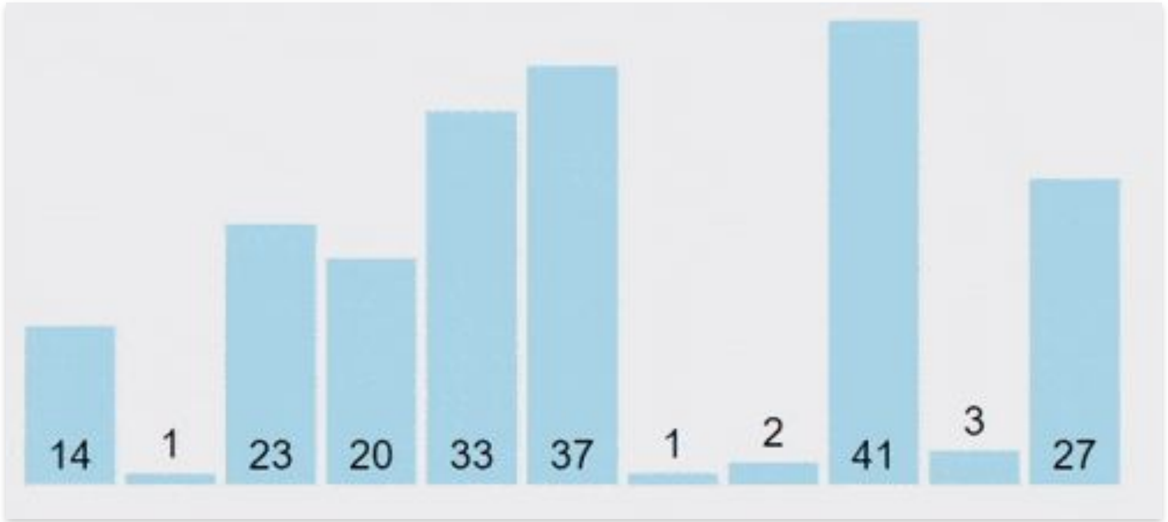
Time & Space Complexity

Time complexity: **O(n²)**

Space complexity: **O(1)**



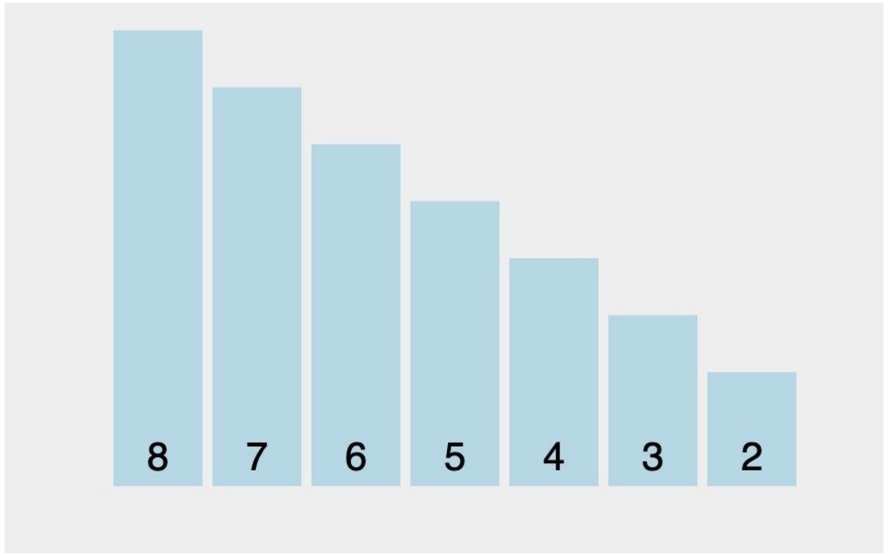
Selection Sort



Time & Space Complexity

Time complexity ? ____________

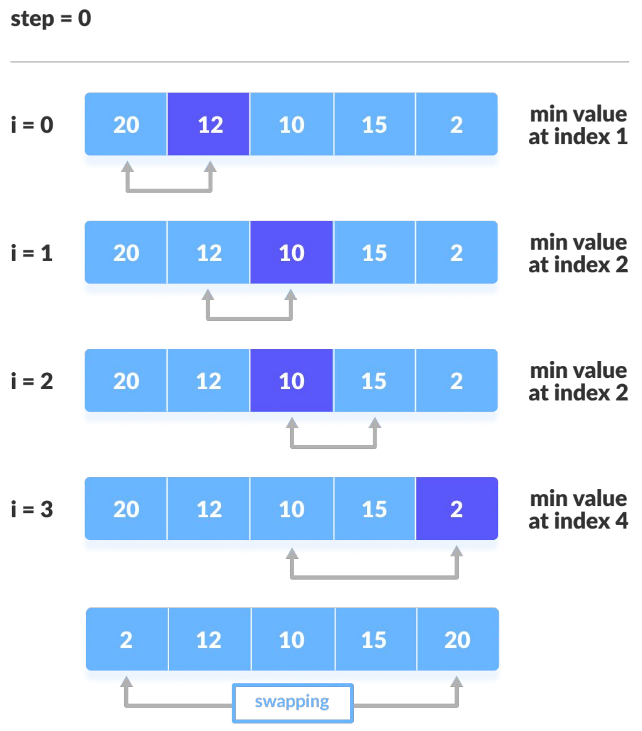
Space complexity ?____________



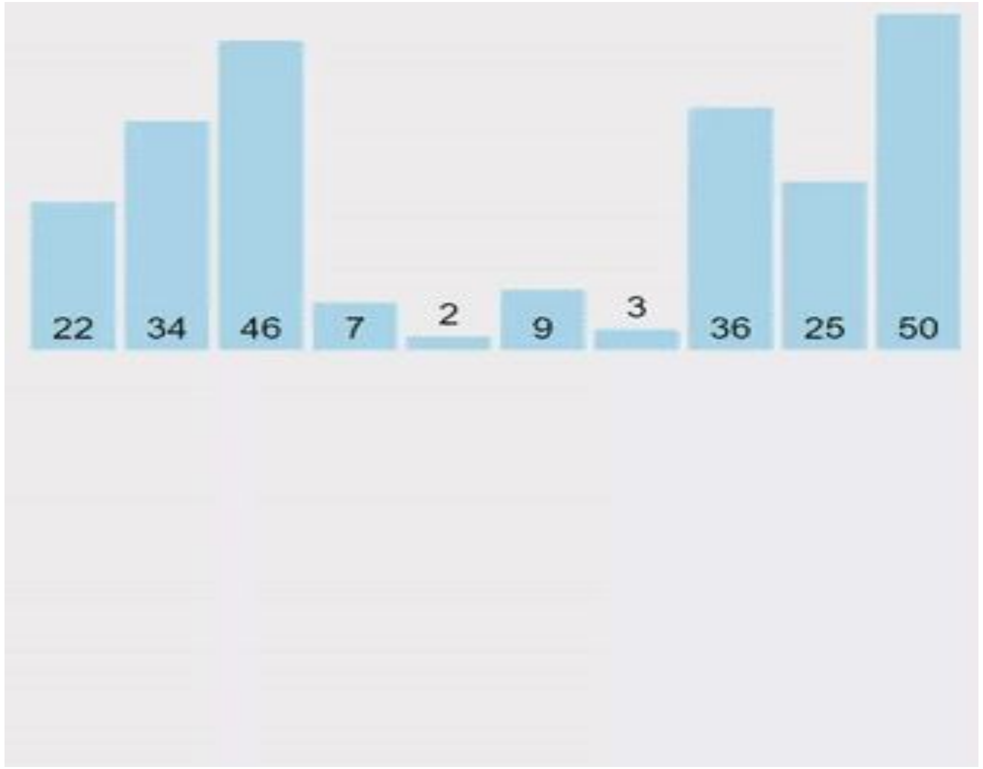
Time & Space Complexity

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

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



Insertion Sort

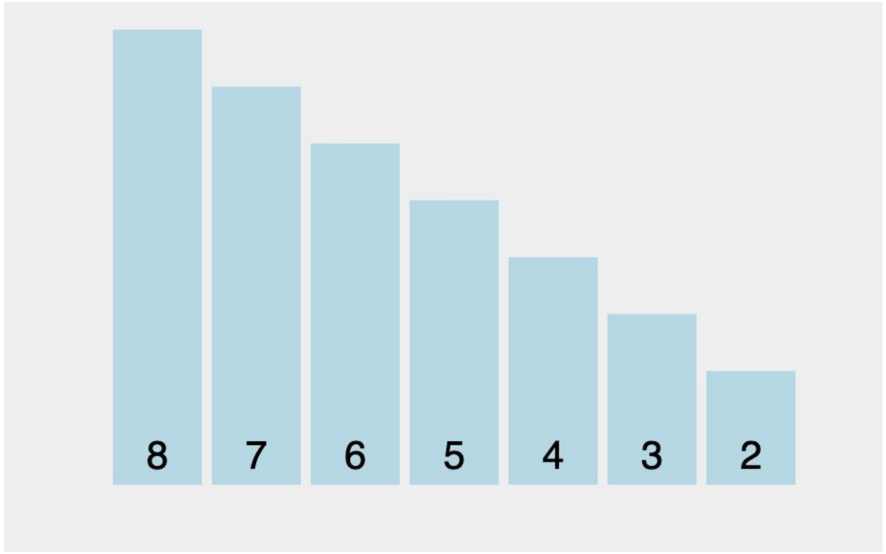


Time & Space Complexity

Worst case ? ____________

Best case ? ____________

Average case ? ____________

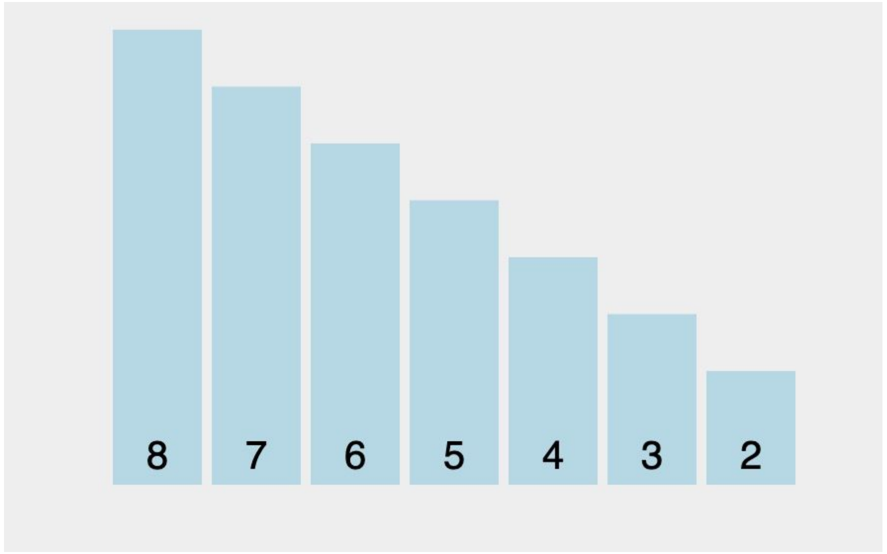


Time & Space Complexity

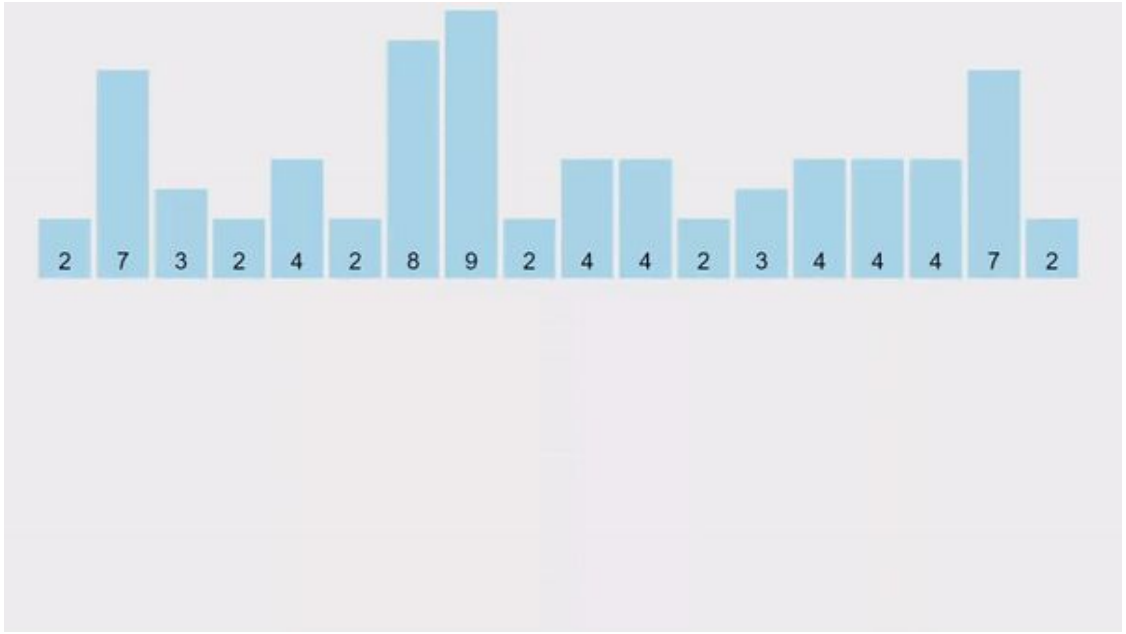
Time complexity: **$O(n^2)$**

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

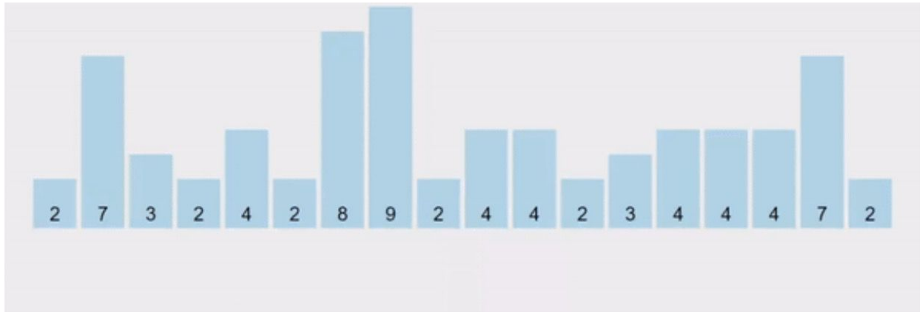
Worst case	**$O(n^2)$**
Best case	**$O(n)$**
Average case	**$O(n^2)$**



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	**O(n+k)**
Best case	**O(n+k)**
Average case	**O(n+k)**

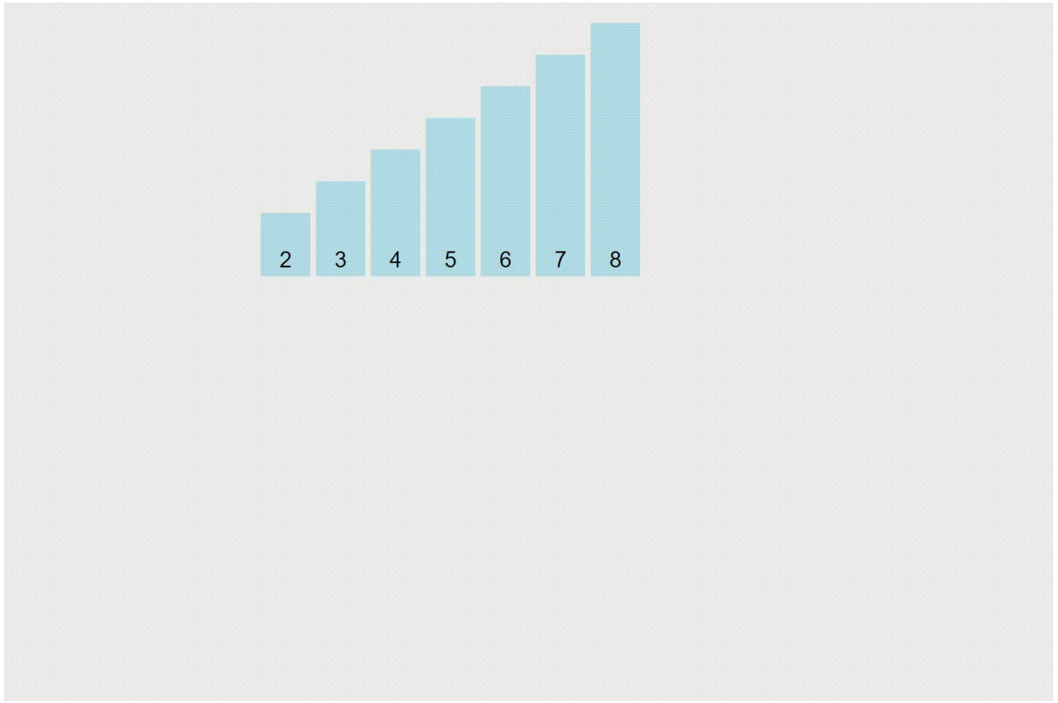
Note: Counting sort is most efficient if the range of input values is not greater than the number of values to be sorted.

Time to get efficient !

Part I

Merge Sort

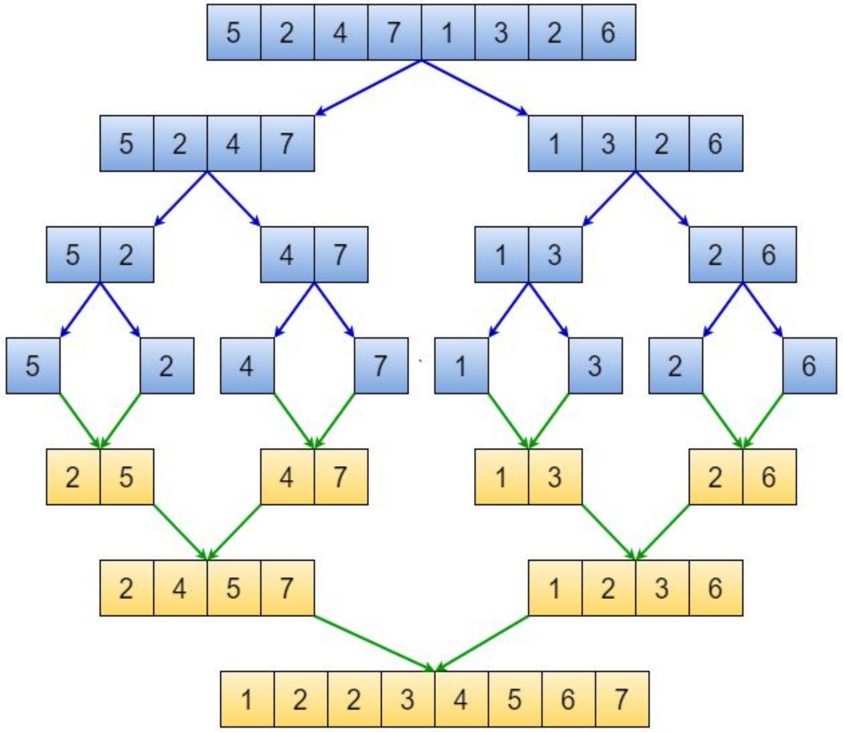
Merge Sort



visualization from: VisuAlgo

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.

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

Implementation

```
def merge(left_half, right_half):
    left_index = 0
    right_index = 0
    sorted_subarray = []

    while left_index < len(left_half) and right_index < len(right_half):
        if left_half[left_index] <= right_half[right_index]:
            sorted_subarray.append(left_half[left_index])
            left_index += 1
        else:
            sorted_subarray.append(right_half[right_index])
            right_index += 1

    sorted_subarray.extend(left_half[left_index:])
    sorted_subarray.extend(right_half[right_index:])

    return sorted_subarray
```

Implementation

```
def mergeSort(left, right, arr):
    if left == right:
        return [arr[left]]
    mid = left + (right - left) // 2
    left_half = mergeSort(left, mid, arr)
    right_half = mergeSort(mid + 1, right, arr)

    return merge(left_half, right_half)
```

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

?

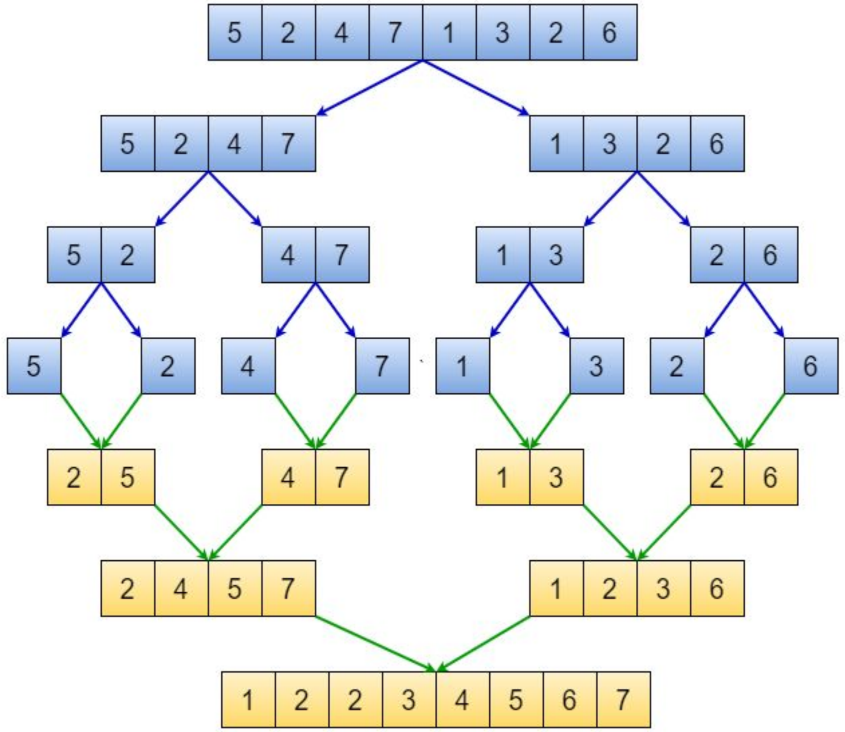
Q: What do you think is the time complexity for the aforementioned sorting Algorithm?

Time & Space Complexity

Worst case ________

Best case ________

Average case ________

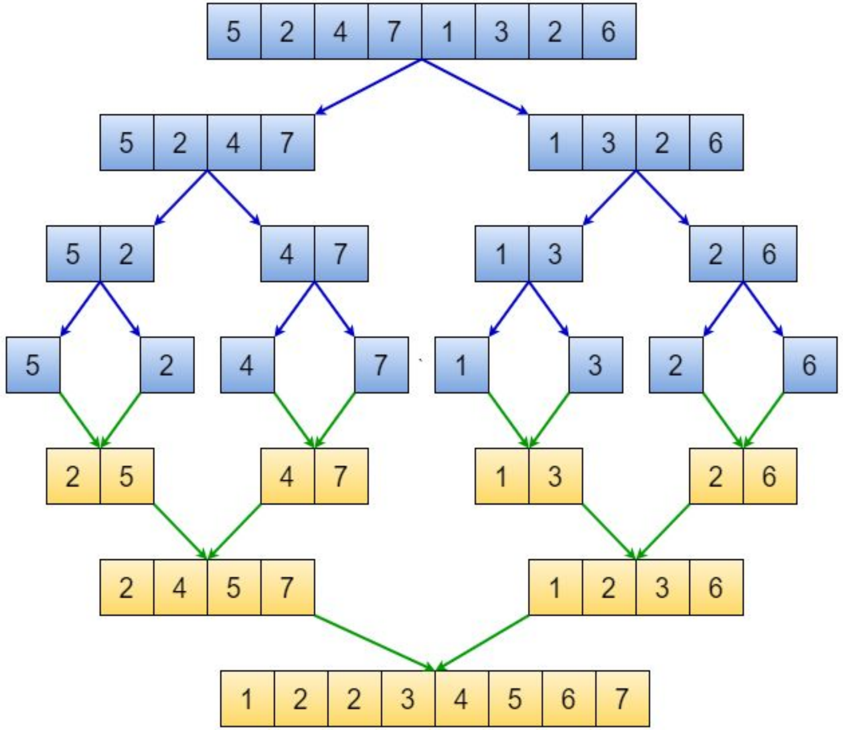


Time & Space Complexity

Time complexity: **O(nlogn)**

Space complexity: **O(n)**

Worst case	**O(nlogn)**
Best case	**O(nlogn)**
Average case	**O(nlogn)**
Stable	**YES**
In-place	**NO**

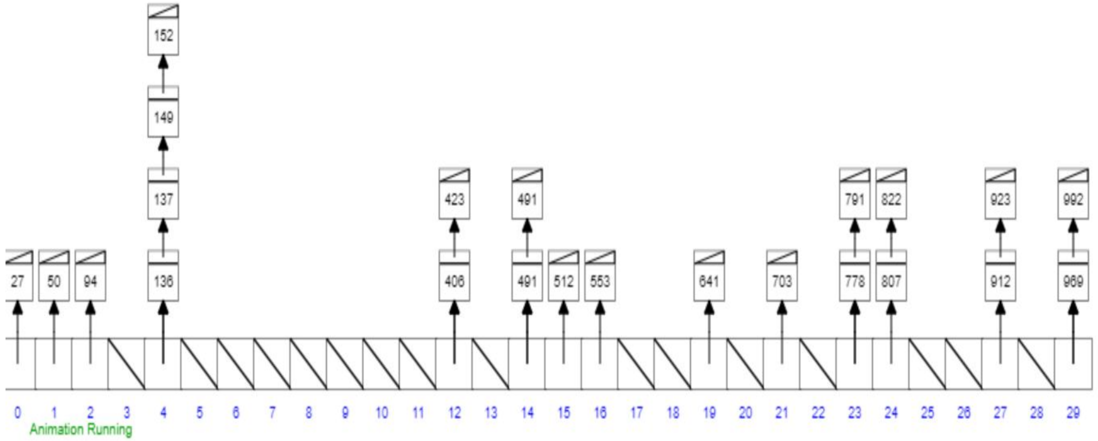


Pair Programming

[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?

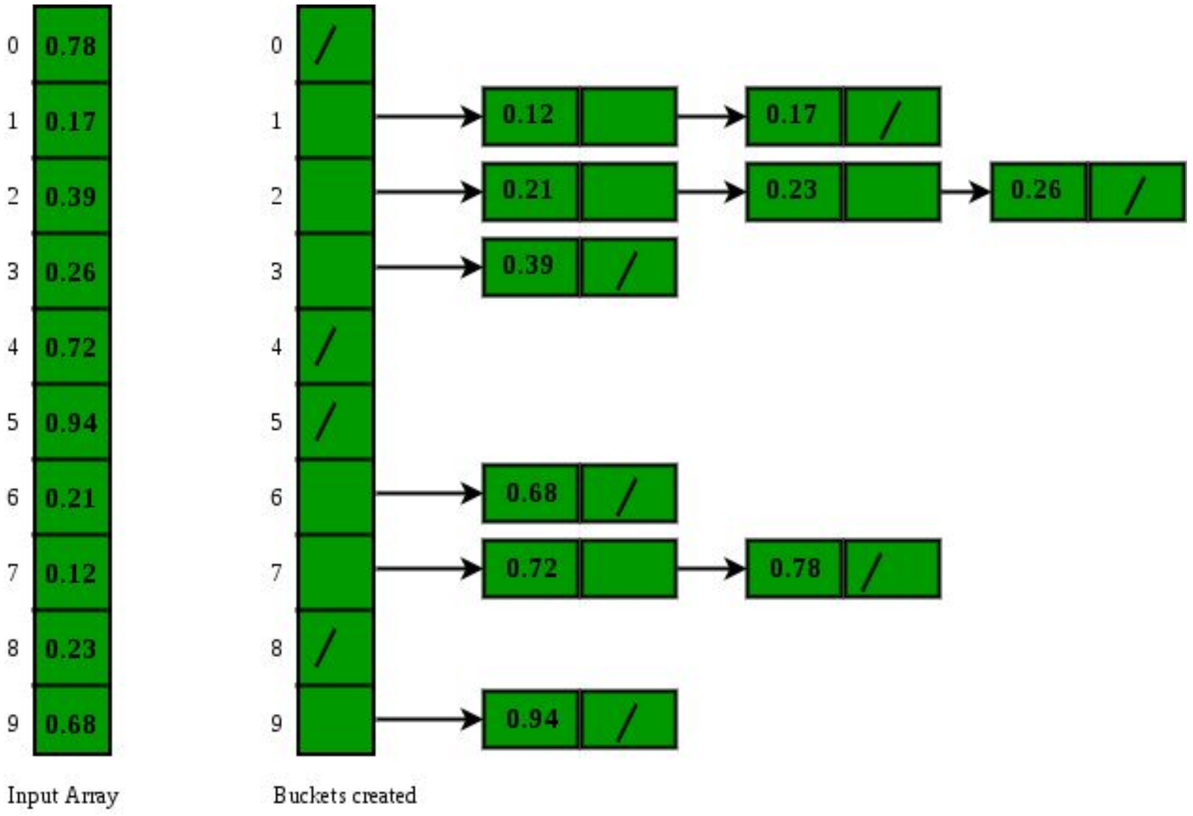
Bucket Sort

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
1
2
3
4
5
6
7
8
9
0.78
0.17
0.39
0.26
0.72
0.94
0.21
0.12
0.23
0.68
Input Array
0.12
0.17
0.21
0.23
0.26
0.39
0.68
0.72
0.78
0.94
Buckets created

Visualization Link

Can you implement the function **bucket_sort** ?

Implement Here

Vanilla Implementation

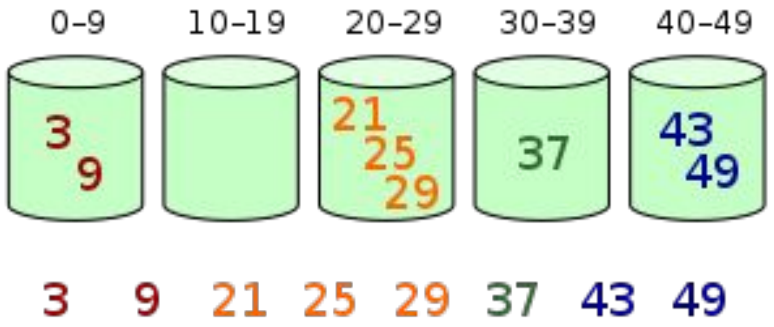
```
def bucketsort(arr, n):
    buckets = [[] for _ in range(n + 1)]
    _min= min(arr)
    ans = []
    _range = max(arr) - _min

    if _range == 0:
        return arr

    for num in arr:
        buckets[int(n*(num - _min) // _range)].append(num)

    for elements in buckets:
        ans.extend(insertion_sort(elements))
    return ans
```

Time & Space Complexity

Worst case ? ____________

Best case ? ____________

Average case ? ____________

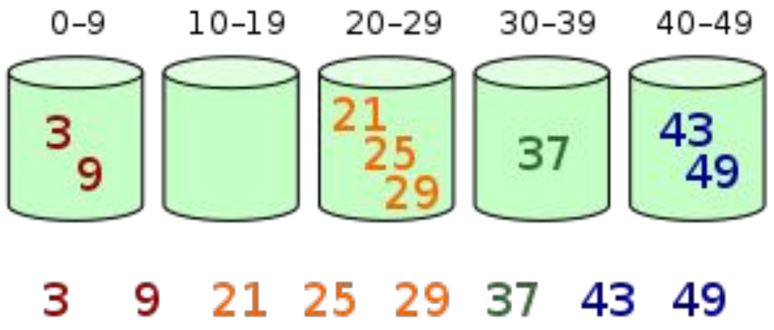


Time & Space Complexity

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**