Unit IV:
Introduction to Computational Problems & Algorithms

Programming Methodology (CSF101)

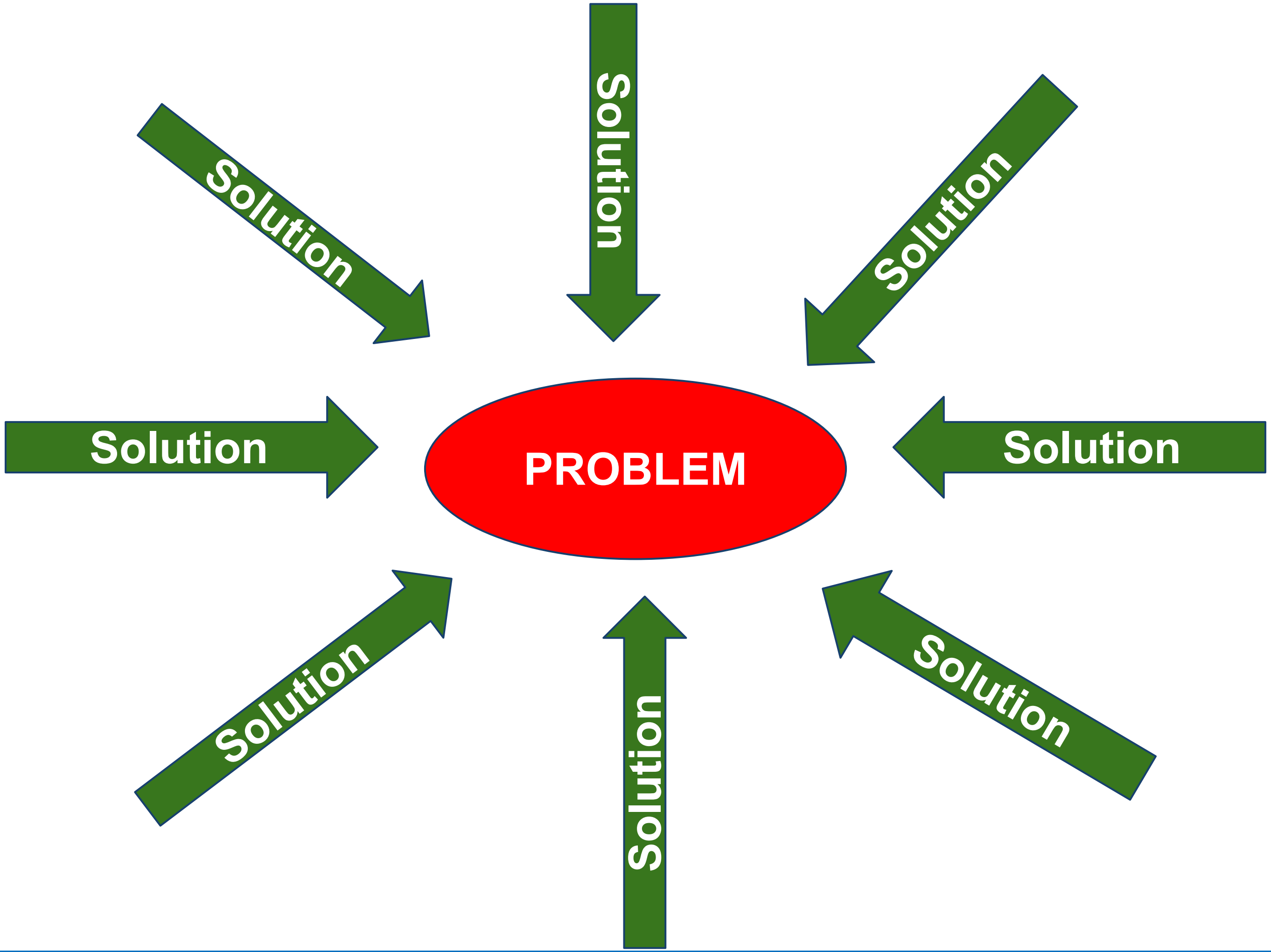
College of Science and Technology

Royal University of Bhutan

Outline

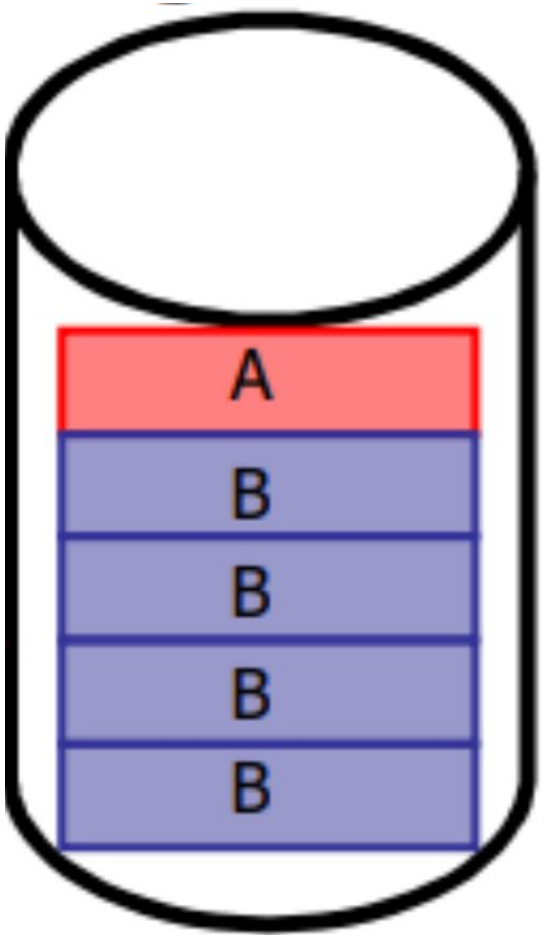
- Space & Time Complexity, Asymptotic Notation
- Sorting Algorithms

Complexity Analysis



Two types of complexity analysis

1. Time Complexity
2. Space Complexity

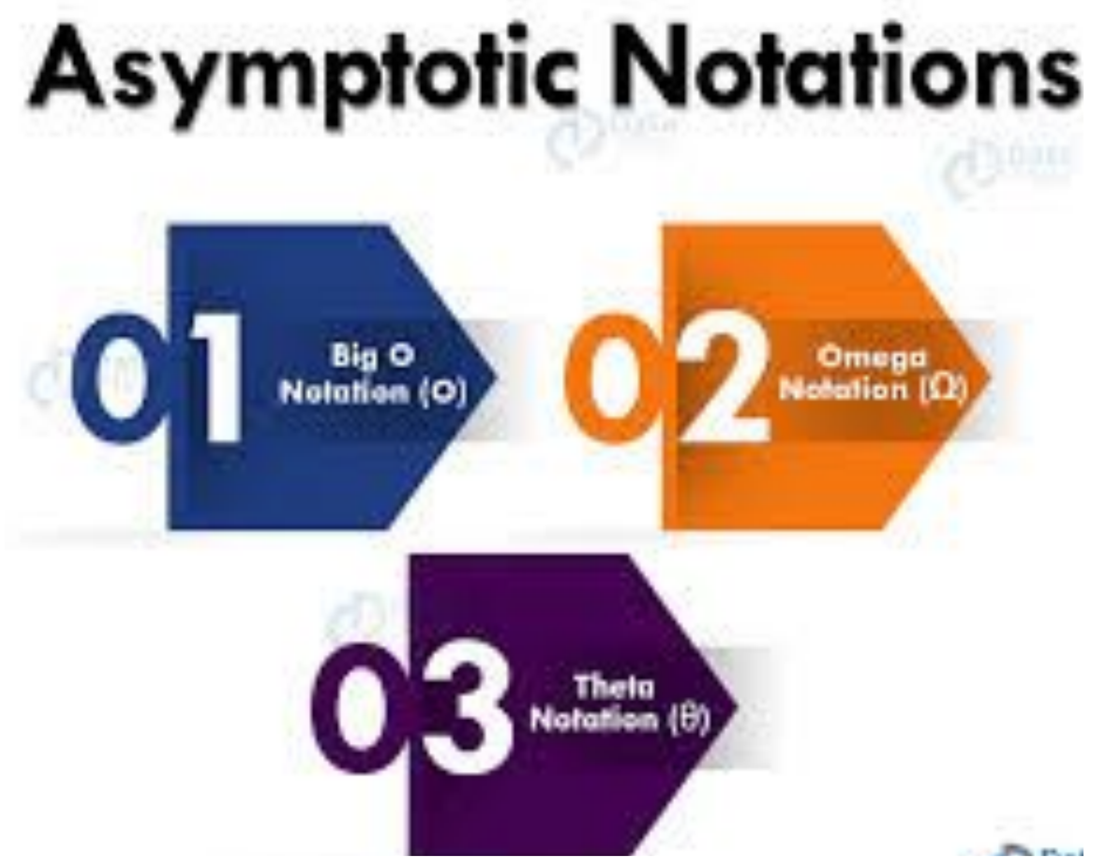


Cont…

$1 < \log n < n^{(1/2)} < n < n^2 < \ldots < n^n$

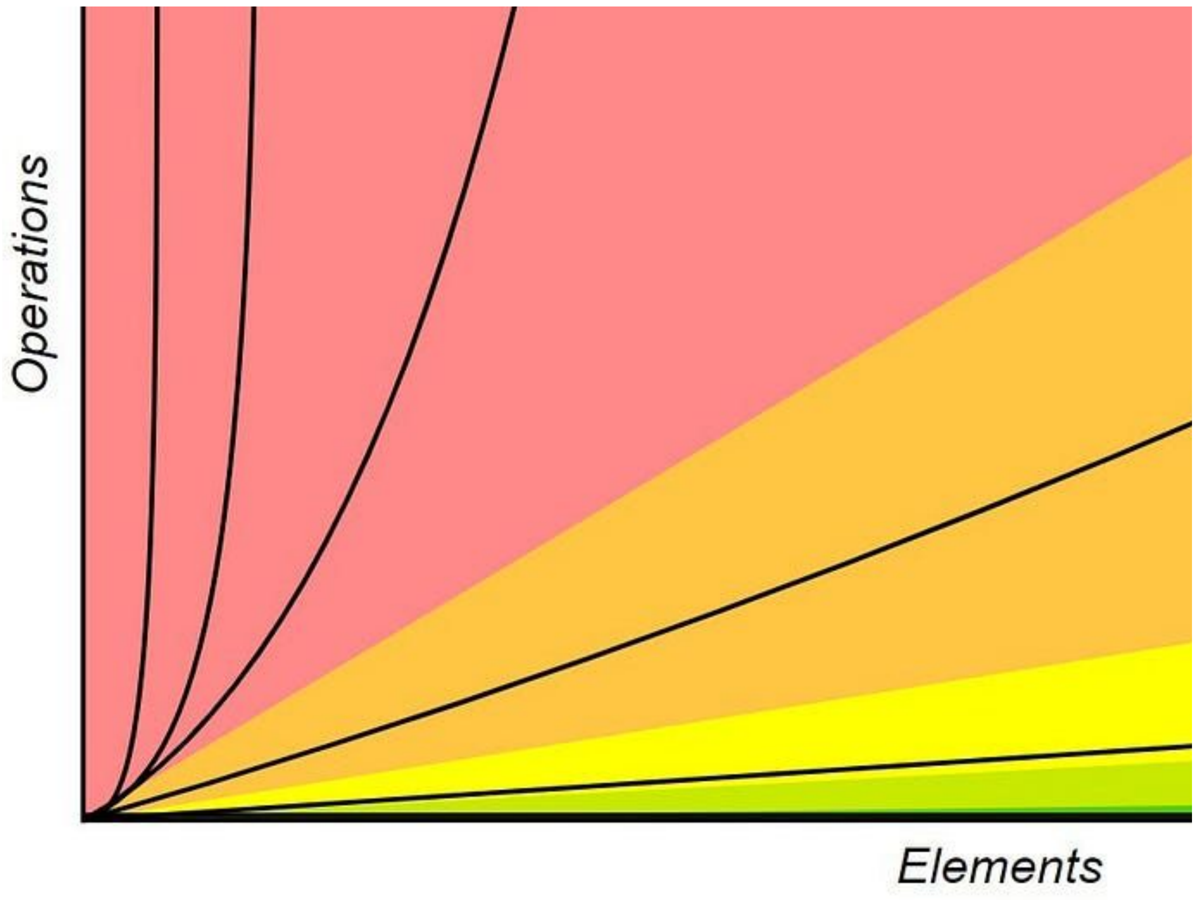
Time in Seconds ??

Space in Mbs??



Run Time Complexity

- Run time is calculated only for executable statements and **NOT** for declarative statements.



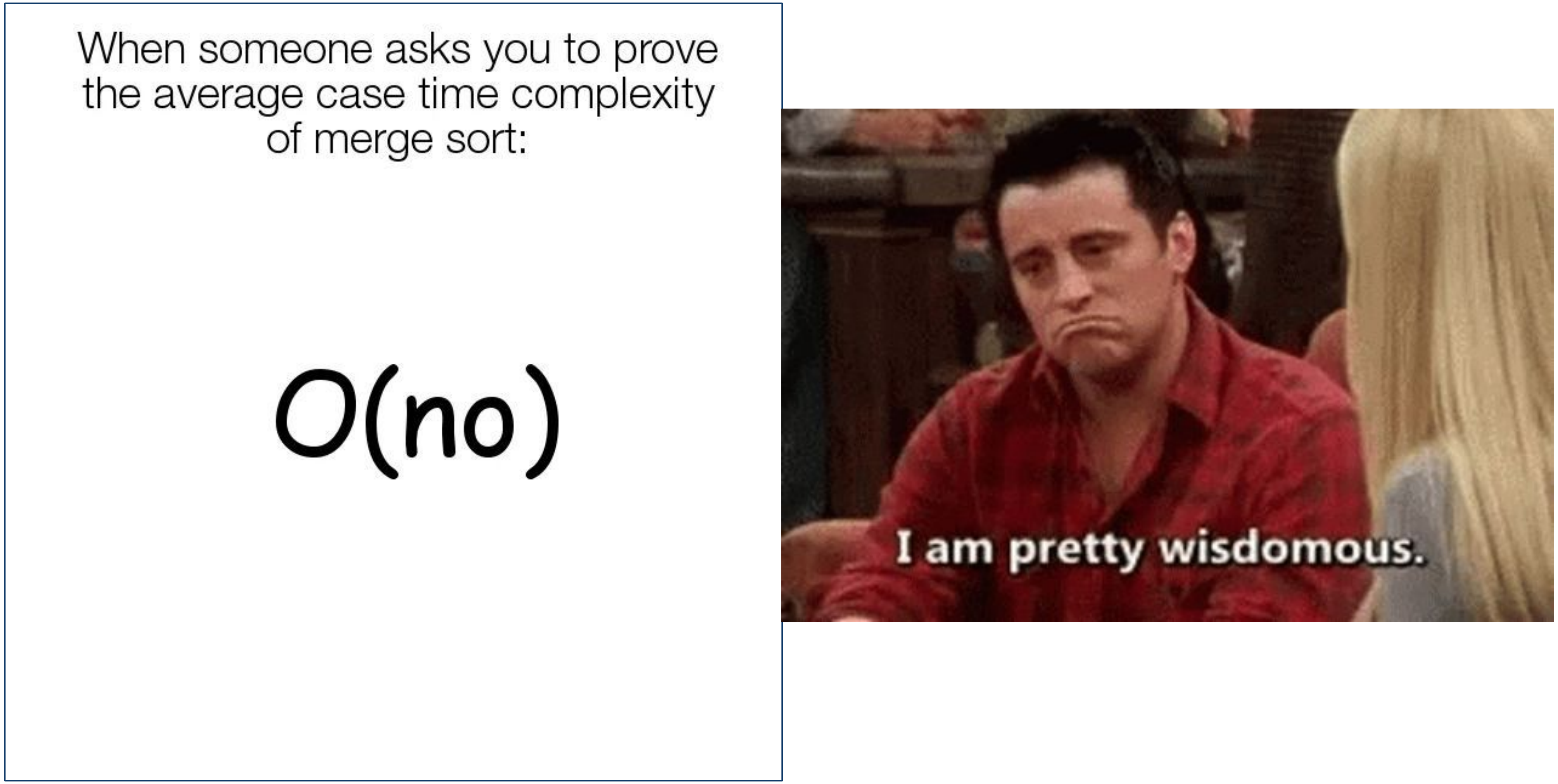
Factors affecting time complexity

- Operations
- Function Calls
- Loops
- Pointer Recursion
- Comparisons

Example

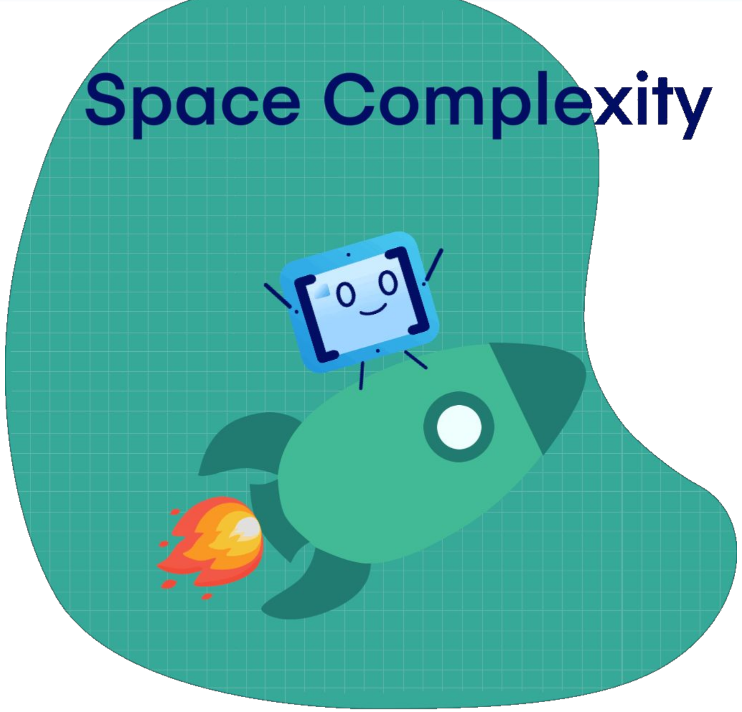
```python
def sum_elements(arr):
    total = 0  # O(1) - Constant time complexity. Initializing a variable requires a fixed amount of time.
    for num in arr:
        total += num  # O(1) - Constant time complexity. Adding a number to 'total' takes a fixed amount of
        time.
    return total  # O(1) - Constant time complexity. Returning the final sum does not depend on the size of the
    input.

# Example usage:
my_list = list(map(int, input("Enter the numbers separated by whitespace: ").split()))  # O(n) - Linear time
complexity. Splitting the input string and converting each element to an integer.
print(sum_elements(my_list))  # Output: sum of elements entered by the user
```


When someone asks you to prove the average case time complexity of merge sort:
O(no)
I am pretty wisdomous.

Space Complexity

Space Complexity = Auxiliary Space + Space used for input values



Factors affecting space complexity

- Variables
- Data Structures
- Function Calls
- Allocations

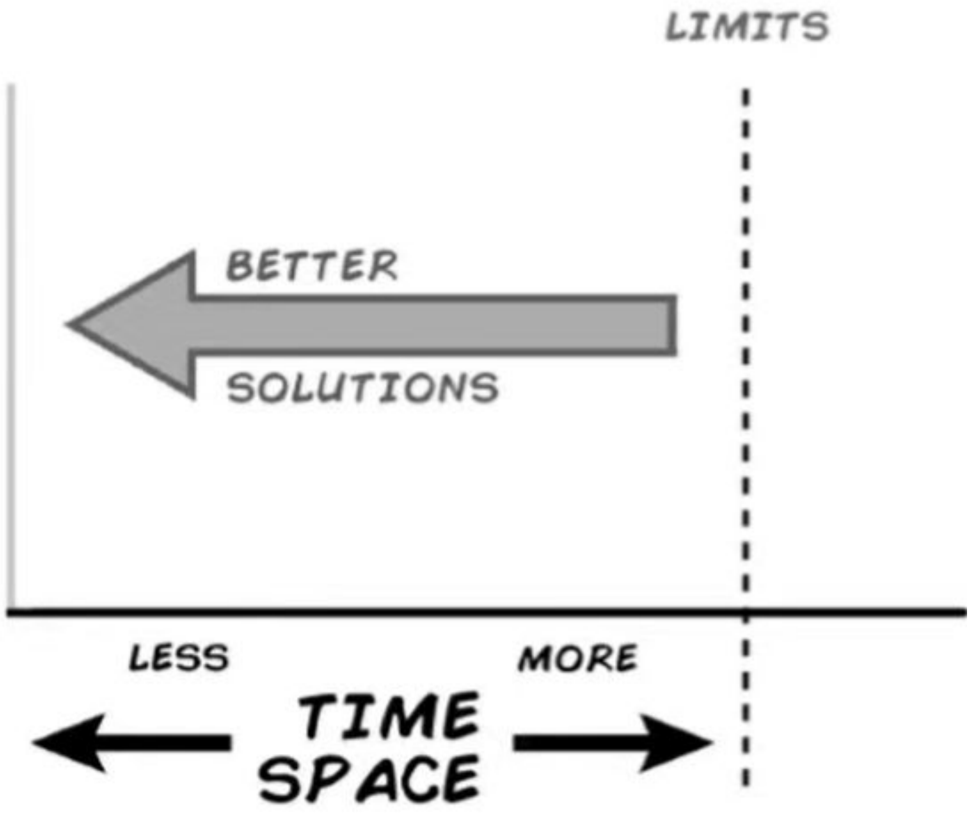
Example

```
def duplicate_elements(arr):
    seen = set()  # O(n) - Linear space complexity. Creating a set to store seen elements.
    duplicates = []  # O(n) - Linear space complexity. Creating a list to store duplicate elements.

    for num in arr:  # O(n) - Linear space complexity. Iterating through each element of the input list.
        if num in seen:  # O(1) - Constant space complexity. Checking if the element has been seen before.
            duplicates.append(num)  # O(1) - Constant space complexity. Appending the duplicate element to the
            list.
        else:
            seen.add(num)  # O(1) - Constant space complexity. Adding the element to the set of seen elements.

    return duplicates  # O(n) - Linear space complexity. Returning the list of duplicate elements.
```

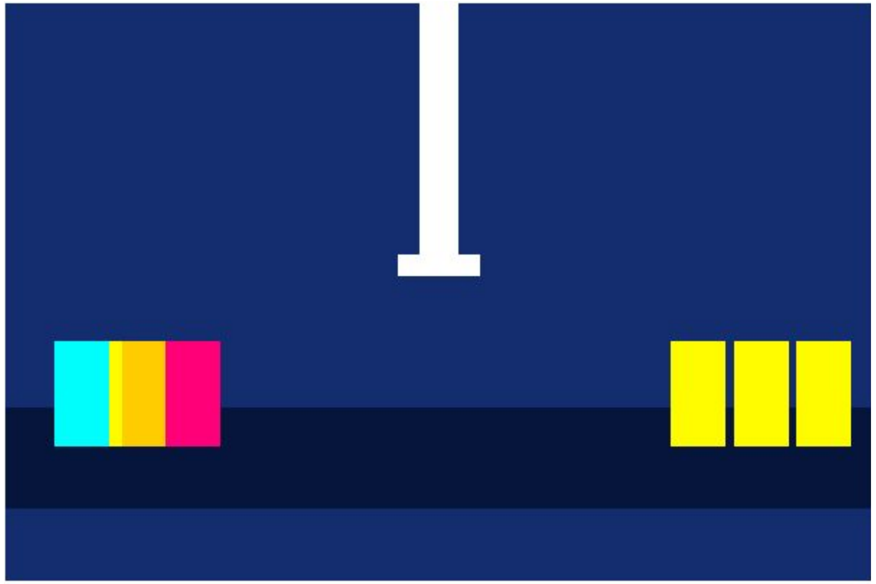
Better Algorithm



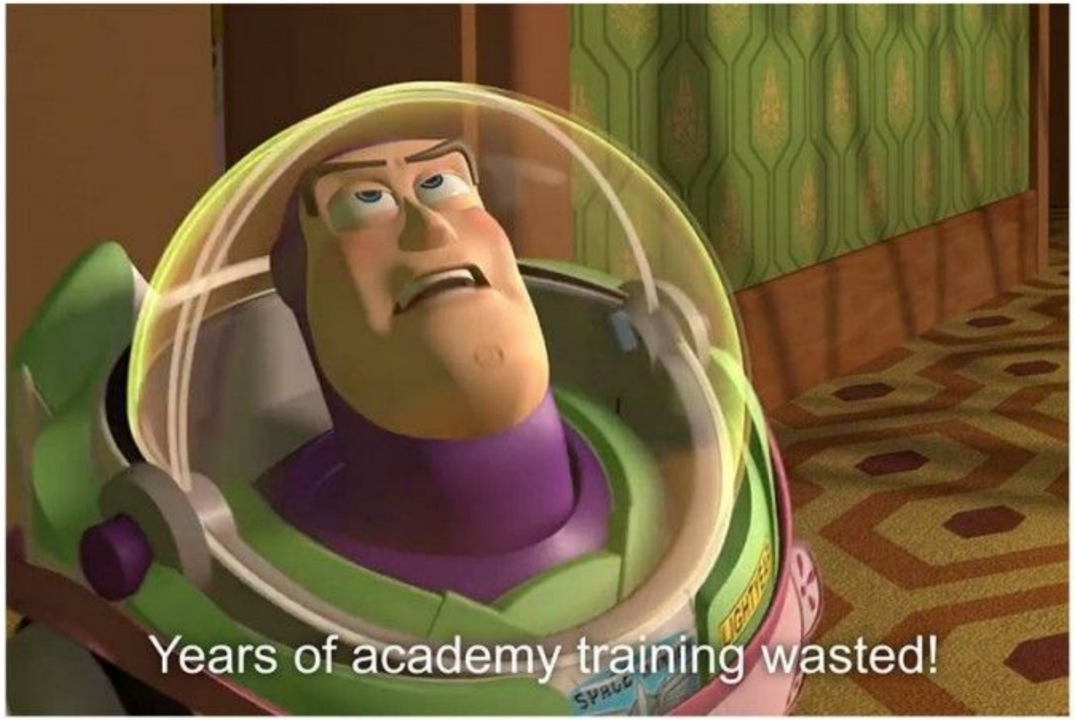
Sorting Algorithm

Some of the importance of sorting are as follows:

1. Searching is made faster
2. Helps to locate various patterns in data
3. Can easily track duplicate values



when there is inbuilt functions for sorting and searching algorithms



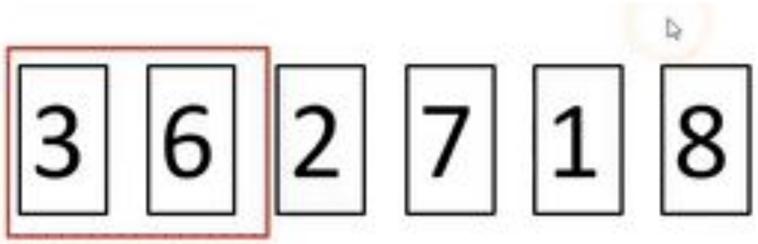
Some of the sorting algorithms

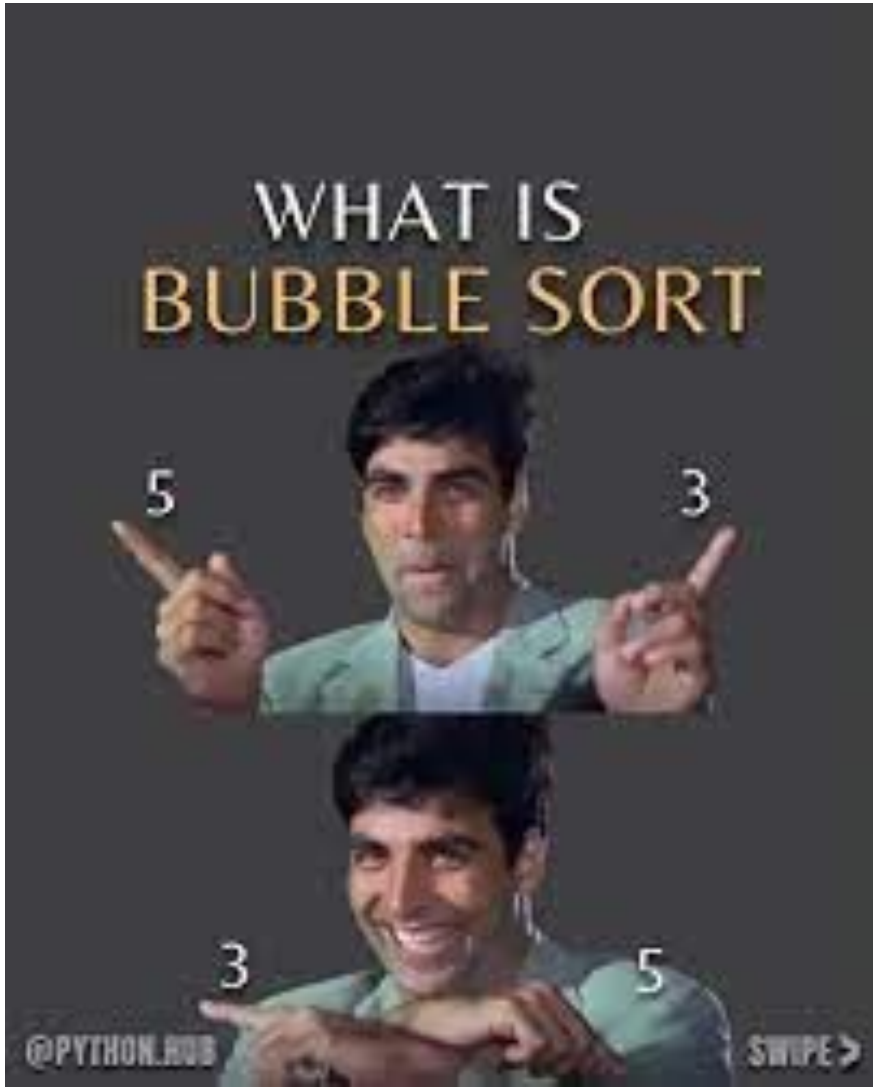
Bubble Sort

Quick Sort

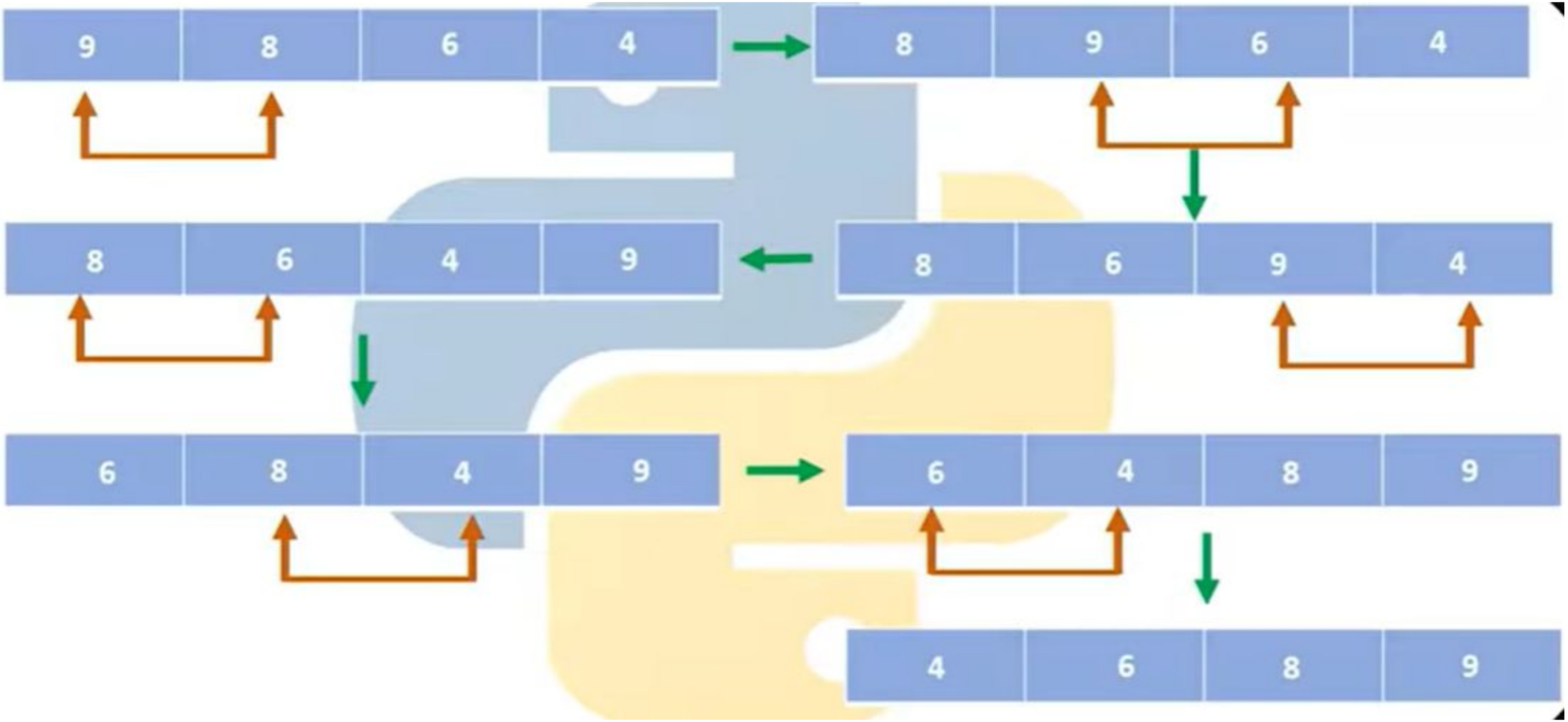
Insertion Sort

Bubble Sort





Example



Cont…

```python
def bubbleSort(arr):
    for i in range(len(arr)):
        for j in range(0, len(arr) - i - 1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                print(f"i: {i}, j: {j} arr: {arr} ")
    return arr

arr = [9, 8, 6, 4]
print(bubbleSort(arr))
```

```
i: 0, j: 0 arr: [8, 9, 6, 4]
i: 0, j: 1 arr: [8, 6, 9, 4]
i: 0, j: 2 arr: [8, 6, 4, 9]
i: 1, j: 0 arr: [6, 8, 4, 9]
i: 1, j: 1 arr: [6, 4, 8, 9]
i: 2, j: 0 arr: [4, 6, 8, 9]
[4, 6, 8, 9]
```

Complexity analysis of bubble sort

TIme Complexity:

Best case: O(n)

Worst Case: O(n^2)

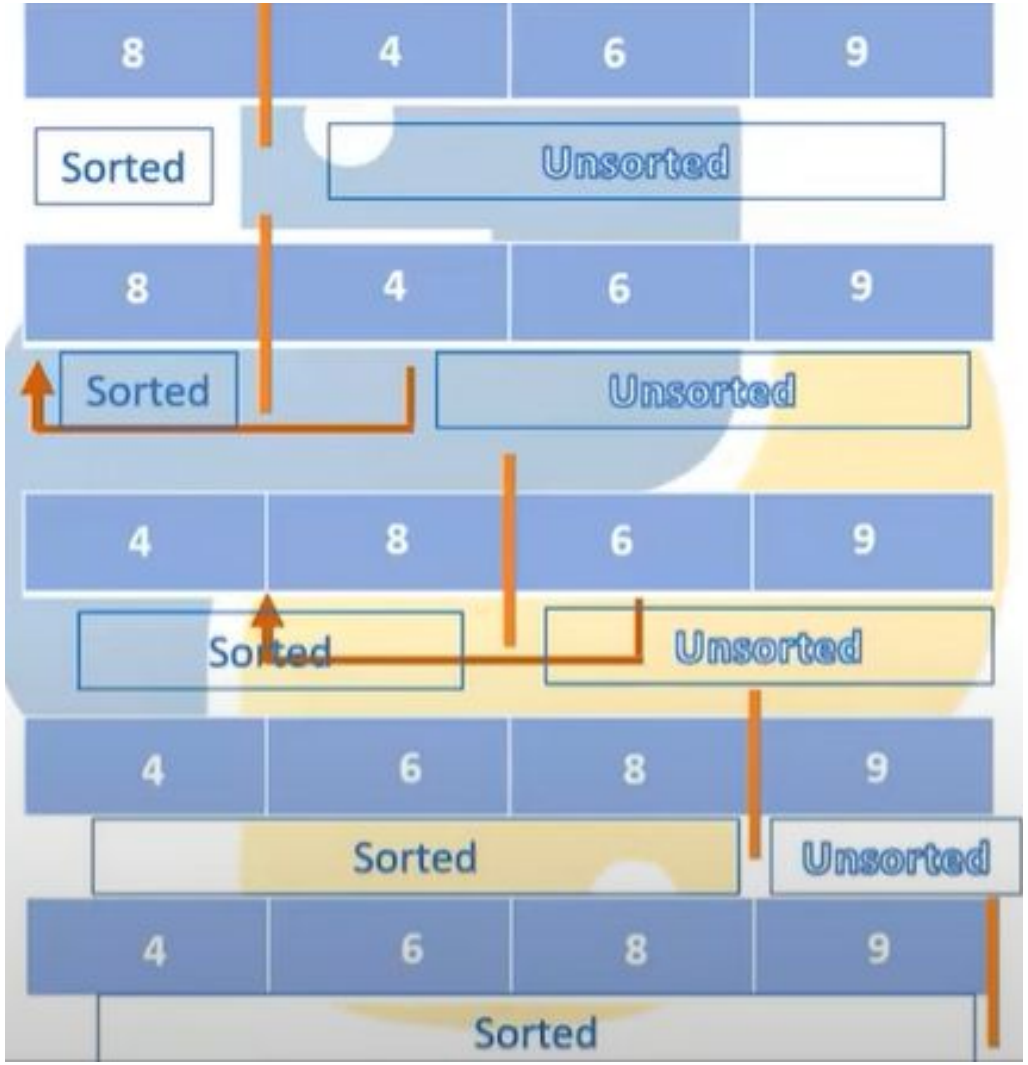
Average Case : O(n^2)

Space Complexity:

O(1)

Insertion Sort

Example


8
4
6
9
Sorted
Unsorted
8
4
6
9
Sorted
Unsorted
4
8
6
9
Sorted
Unsorted
4
6
8
9
Sorted
Unsorted
4
6
8
9
Sorted

Cont…

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        j = i - 1
        while j >= 0 and arr[j+1] < arr[j]:
            arr[j + 1], arr[j] = arr[j], arr[j + 1]
            print(f"i: {i}, j: {j} arr: {arr}"  )
            j -= 1
    return arr

print(insertion_sort([8, 4, 6, 9]))
```

```
i: 1, j: 0 arr: [4, 8, 6, 9]
i: 2, j: 1 arr: [4, 6, 8, 9]
[4, 6, 8, 9]
```

Complexity Analysis of Insertion Sort

Time Complexity:

Best Case : O(n)

Worst Case : O(n^2)

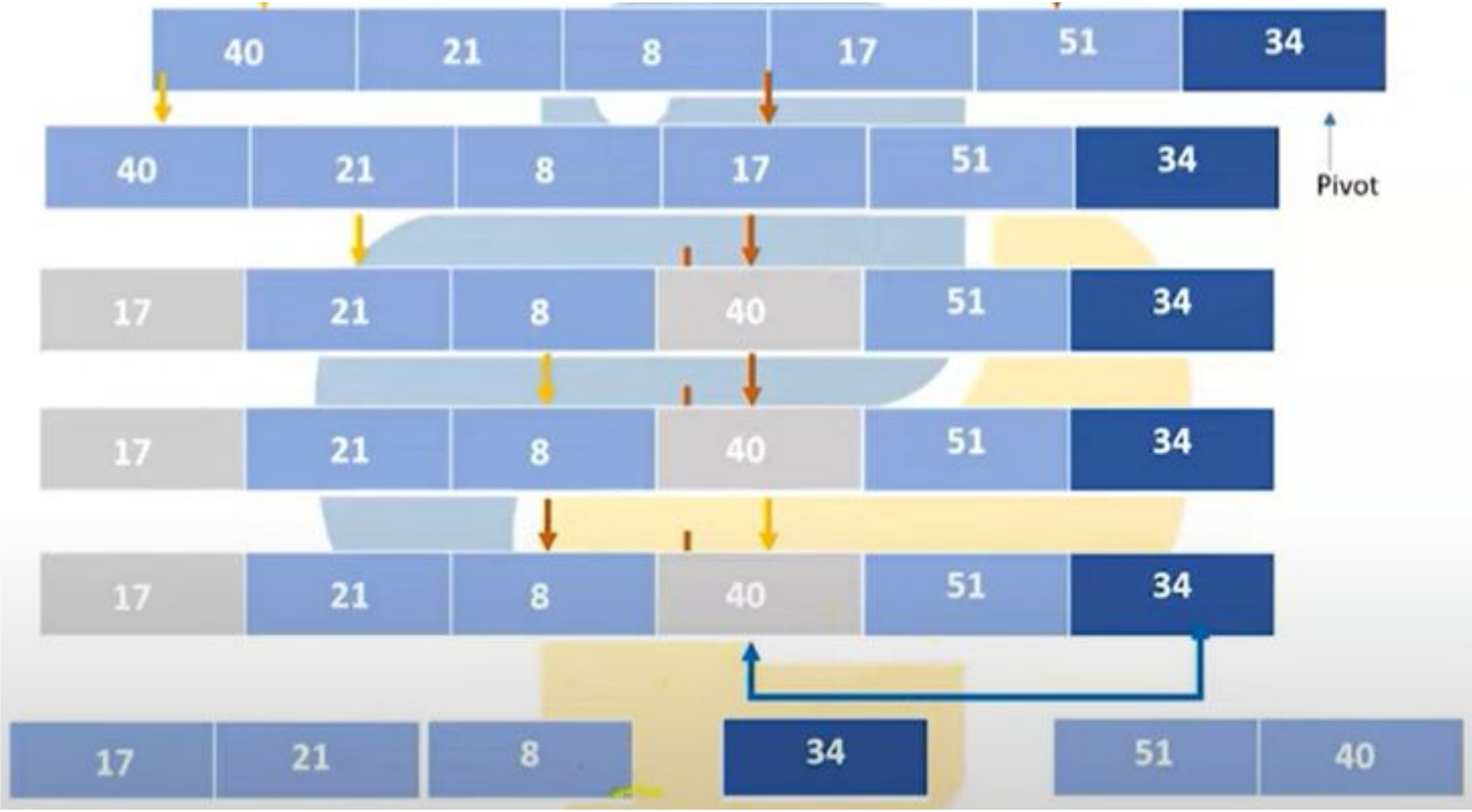
Average Case: O(n^2)

Space Complexity:

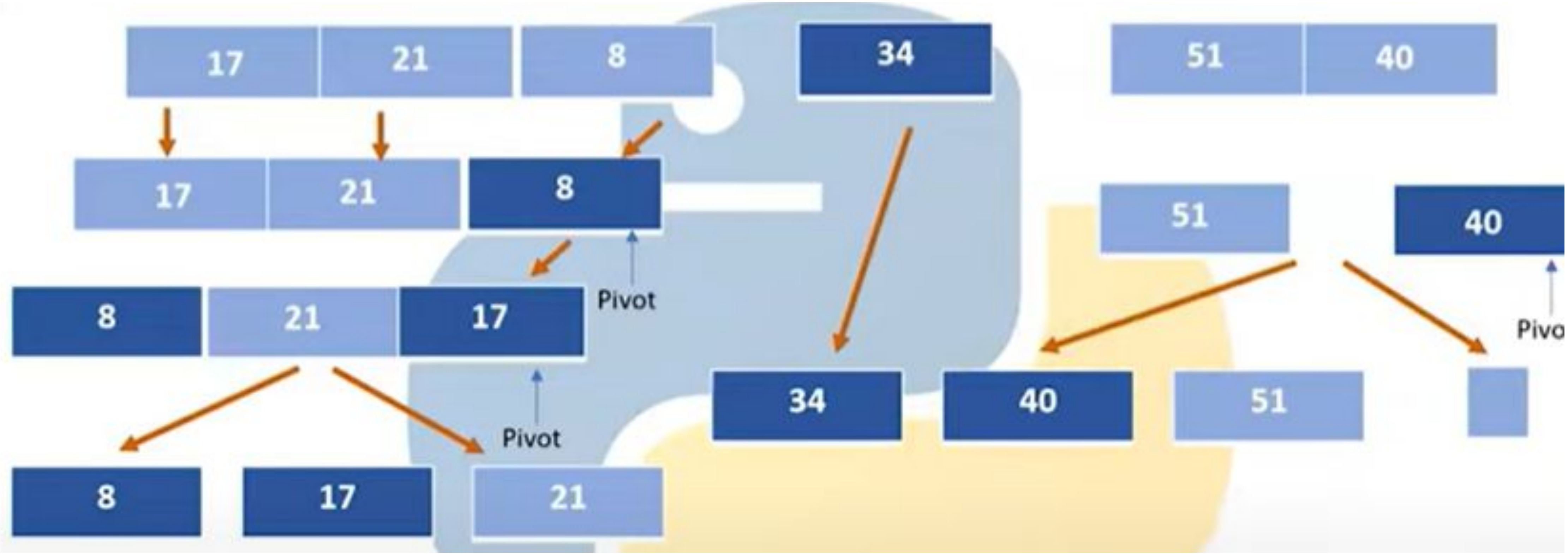
O(1)

Quick Sort

Example


40
21
8
17
51
34
40
21
8
17
51
34
Pivot
17
21
8
40
51
34
17
21
8
40
51
34
17
21
8
40
51
34
17
21
8
34
51
40

Cont…


17
21
8
34
51
40
17
21
8
51
40
Pivot
8
21
17
Pivo
Pivot
34
40
51
8
17
21

Cont…

```python
def quick_sort(arr):
    # Base case: If the length of the list is 0 or 1, it is already sorted.
    if len(arr) <= 1:
        return arr
    else:
        # Choosing the pivot as the last element of the list.
        pivot = arr[-1]
        print(f"pivot: {pivot}")

        # Partitioning the list into two sublists:
        # - 'left' contains elements less than or equal to the pivot.
        # - 'right' contains elements greater than the pivot.
        left = [x for x in arr[:-1] if x <= pivot]
        right = [x for x in arr[:-1] if x > pivot]
        print(f"left: {left}, right: {right}")

        # Recursively applying quick_sort to the left and right sublists,
        # and concatenating the sorted sublists with the pivot.
        return quick_sort(left) + [pivot] + quick_sort(right)

# Example usage:
my_list = [40, 21, 8, 17, 51, 34]
sorted_list = quick_sort(my_list)
print("Sorted list:", sorted_list)
```

```
pivot: 34
left: [21, 8, 17], right: [40, 51]
pivot: 17
left: [8], right: [21]
pivot: 51
left: [40], right: []
Sorted list: [8, 17, 21, 34, 40, 51]
```

Complexity analysis of Quick Sort

Time Complexity:

Best Case: O(n logn)

Worst Case: O(n^2)

Average Case: O(n logn)

Space Complexity:

O(logn)

Reference

GfG (2023) Time Complexity and Space Complexity, GeeksforGeeks. Available at: https://www.geeksforgeeks.org/time-complexity-and-space-complexity/.

Time and space complexity of sorting algorithms (no date) shiksha. Available at: https://www.shiksha.com/online-courses/articles/time-and-space-complexity-of-sorting-algorithms-blogId-152755.


Thank You