ENG 346
Data Structures and Algorithms for Artificial Intelligence
Sorting

Dr. Mehmet PEKMEZCİ

mpekmezci@gtu.edu.tr

https://github.com/mehmetpekmezci/GTU-ENG-346

ENG-346-FALL-2025 Teams code is **4b108kr**

Agenda

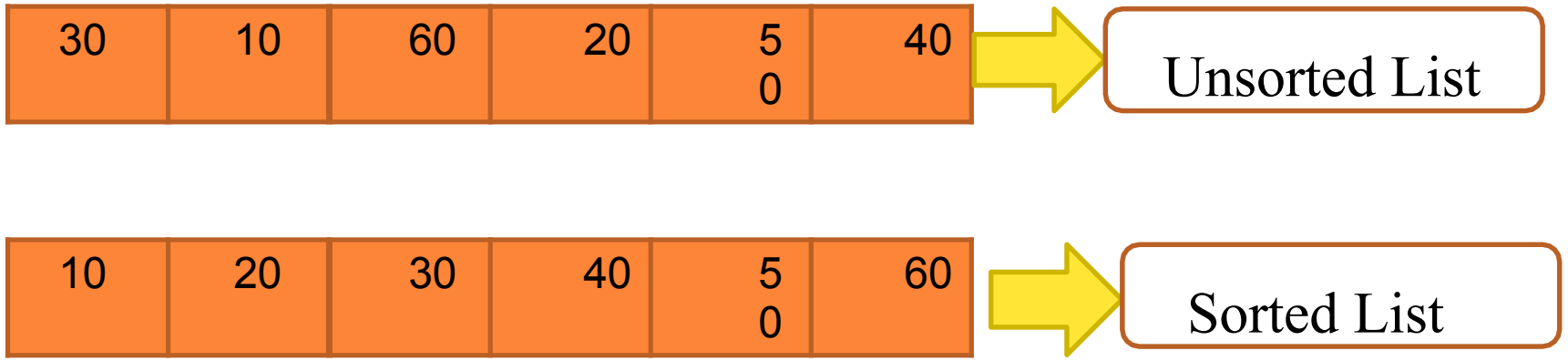
- Bubble Sort
- Selection Sort
- Insertion Sort
- Selection Sort
- Quick Sort
- Merge Sort
- Heap Sort

Complexity

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)

SORTING

- Sorting refers to operations of arranging a set of data in a given order.



Sorting

- Search Operations:
  - Sorted data allows for efficient search operations. Binary search, for example, can be performed on a sorted array or list in O(log n) time, which is significantly faster than linear search on unsorted data (O(n)).
- Insertion and Deletion:
  - In certain data structures like binary search trees, maintaining sorted order simplifies and accelerates the process of insertion and deletion. It ensures that the tree remains balanced and allows for faster search operations.
- Efficient Merging:
  - When working with data structures like heaps or priority queues, merging two sorted structures becomes a more straightforward task. This is particularly important in applications such as external sorting.
- Range Queries:
  - Sorting facilitates efficient range queries. For example, finding all elements within a given range in a sorted array or list can be done much more efficiently than in an unsorted structure.

Sorting

- Data Retrieval:
  - In scenarios where data needs to be retrieved in a specific order, having the data pre-sorted simplifies and speeds up the retrieval process. This is particularly important in databases and information retrieval systems.
- Optimizing Algorithms:
  - Certain algorithms and data structures perform better or are easier to implement when working with sorted data. For example, dynamic programming algorithms may take advantage of sorted input to optimize their runtime.
- Duplicate Removal:
  - Sorting facilitates the removal of duplicates in a dataset. Adjacent duplicate elements can be easily identified and removed in a single pass through sorted data.
- Intersection of Sets:
  - When working with sets, determining the intersection of two sets becomes more efficient if the sets are sorted. This is especially relevant in applications involving database queries.

Applications of Sorting

- **Search Algorithms:**
  - Efficient searching often relies on pre-sorted data, improving search times.
- **Database Indexing:**
  - Crucial for creating indexes in databases, allowing for faster query operations.
- **E-commerce and Online Marketplaces:**
  - Used to display products in a particular order, such as by price or popularity.
- **Log Analysis:**
  - Sorting log entries by timestamp aids in identifying patterns and analyzing system behavior.
- **Task Scheduling:**
  - Sorting is applied in scheduling tasks to optimize the order in which they are executed.
- ...

Types of Sorting

- Internal Sorting:
  - If all the data to be sorted can be adjusted in main memory then it is called as Internal Sorting.

- External Sorting:
  - If data to be stored is large and requires external memory then the type is called as External Sorting.

Bubble Sort

- Works by repeatedly swapping the adjacent elements if they are in the wrong order.
- Not suitable for large data sets as its average and worst-case time complexity are quite high.

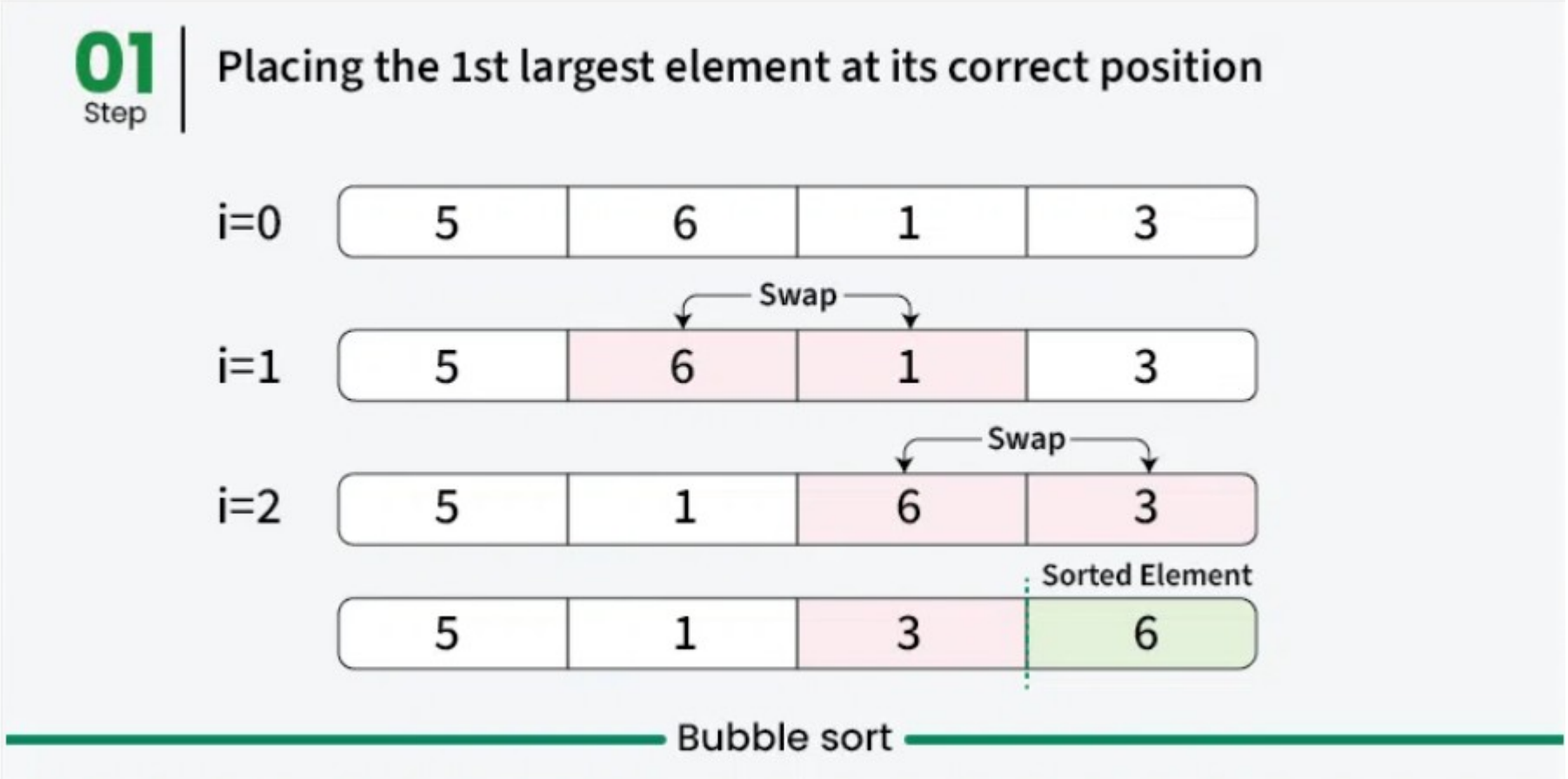
- Data is sorted in multiple passes.
  - After the first pass, the maximum element goes to end (its correct position).
  - After second pass, the second largest element goes to second last position.
  - After k passes, the largest k elements must have been moved to the last k positions.
- In a pass, consider remaining elements and compare all adjacent and swap if larger element is before a smaller element. If we keep doing this, we get the largest (among the remaining elements) at its correct position.

Bubble Sort – algorithm

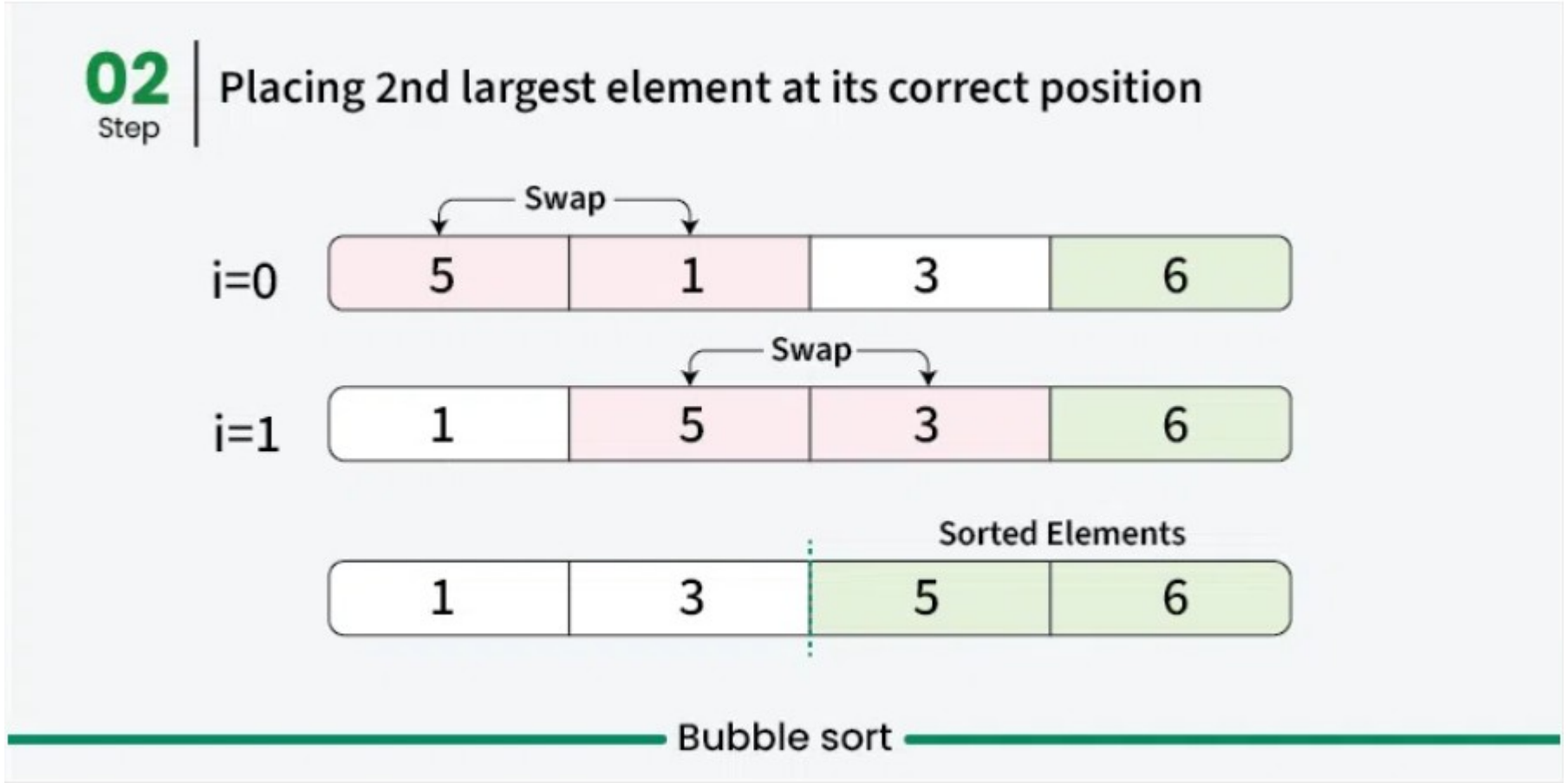
```
def bubbleSort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        swapped = False
        # Last i elements are already in place
        for j in range(0, n-i-1):
            # Traverse the array from 0 to n-i-1
            # Swap if the element is greater than the next
element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if (swapped == False):
            break
```

Time Complexity: $O(n^2)$
Auxiliary Space: $O(1)$

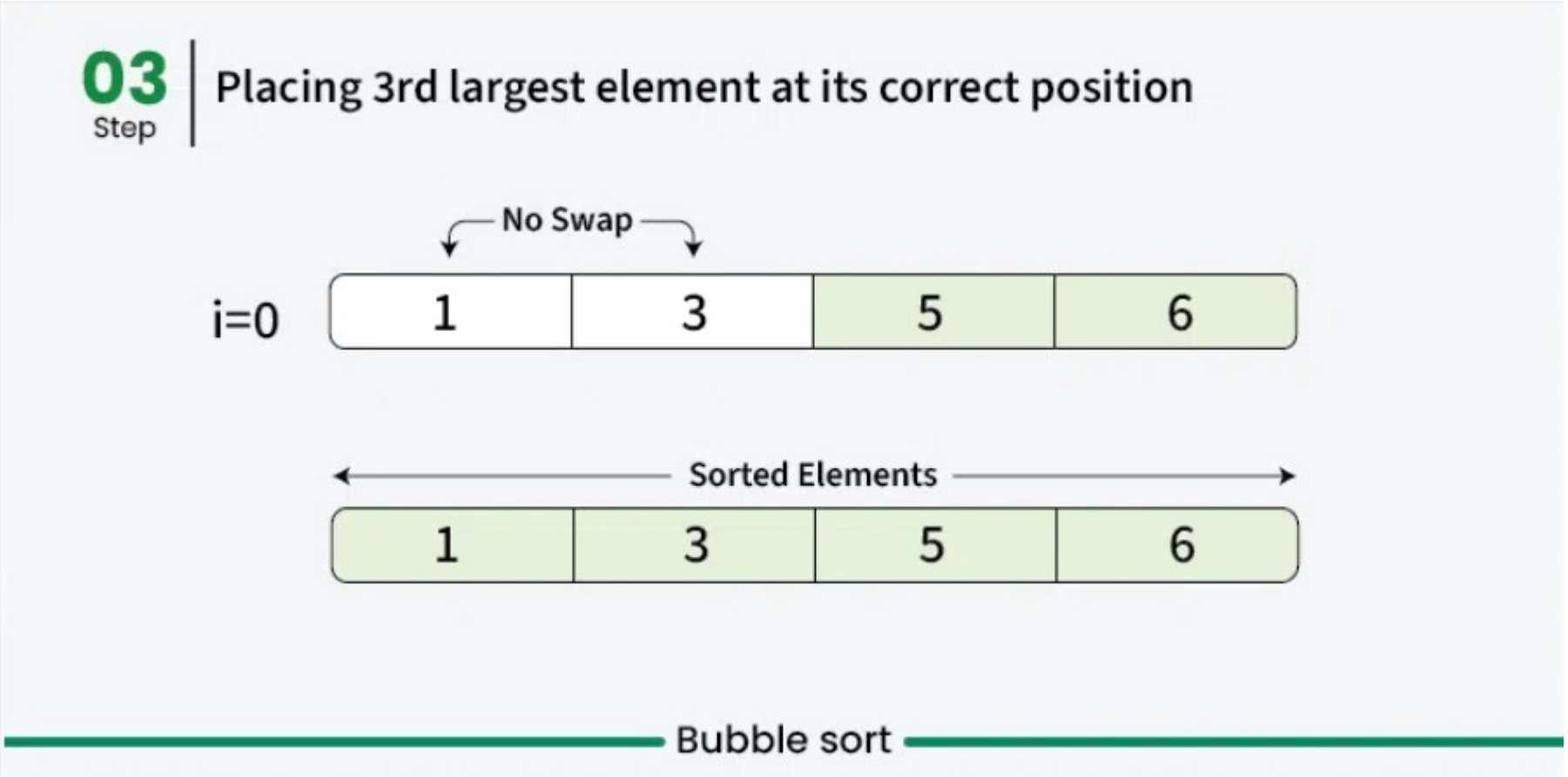
Bubble Sort


01
Step
Placing the 1st largest element at its correct position
i=0
5
6
1
3
Swap
i=1
5
6
1
3
Swap
i=2
5
1
6
3
Sorted Element
5
1
3
6
Bubble sort

Bubble Sort


02
Step
Placing 2nd largest element at its correct position
Swap
i=0
5
1
3
6
Swap
i=1
1
5
3
6
Sorted Elements
1
3
5
6
Bubble sort

Bubble Sort


03
Step
Placing 3rd largest element at its correct position
No Swap
i=0
1
3
5
6
Sorted Elements
1
3
5
6
Bubble sort

Bubble Sort – continued

Advantages of Bubble Sort:

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

Disadvantages of Bubble Sort:

- Bubble sort has a time complexity of $O(n^2)$ which makes it very slow for large data sets.
- Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

Selection Sort

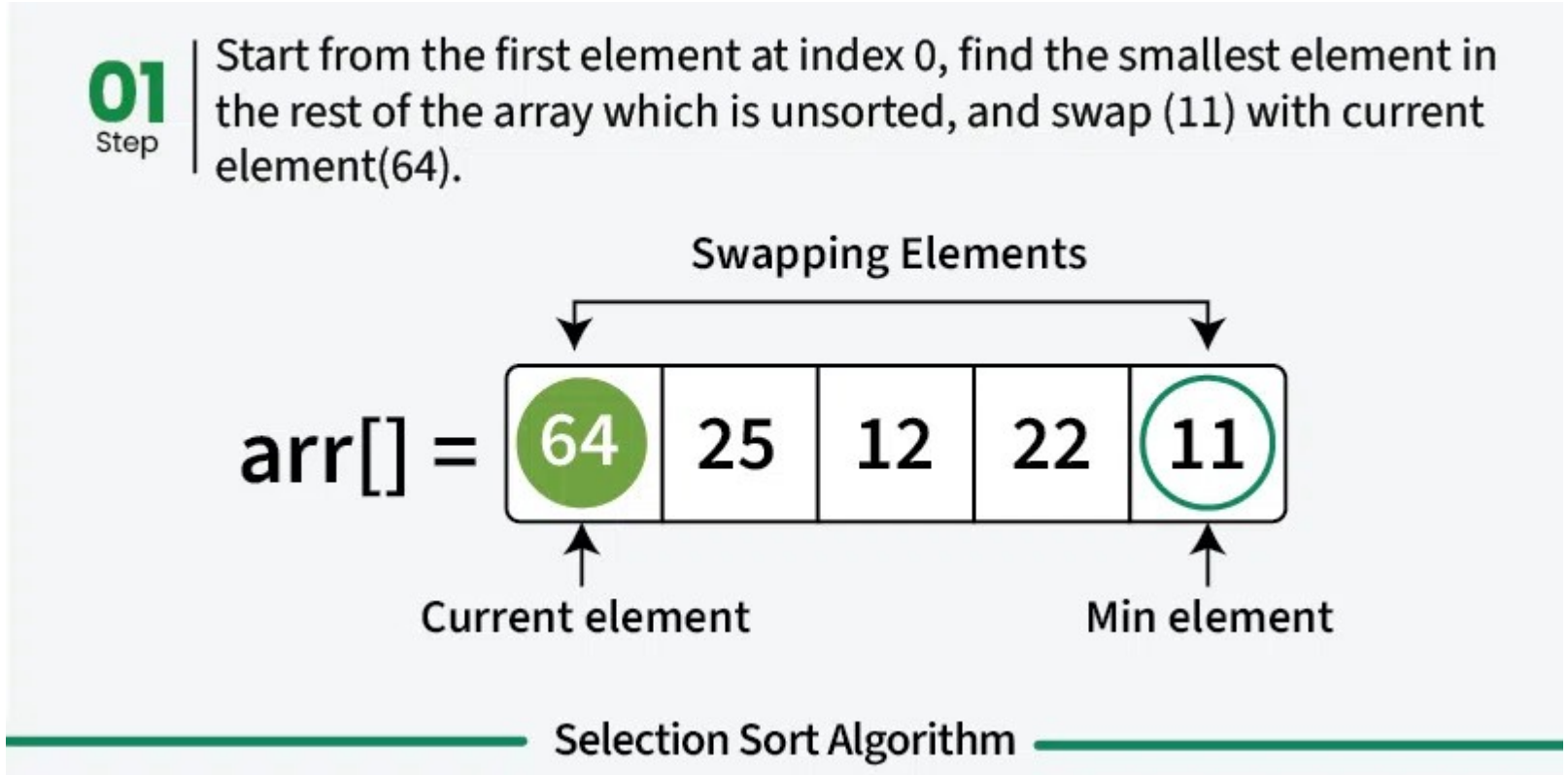
- Comparison-based sorting algorithm.
- Sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element.
- This process continues until the entire array is sorted.

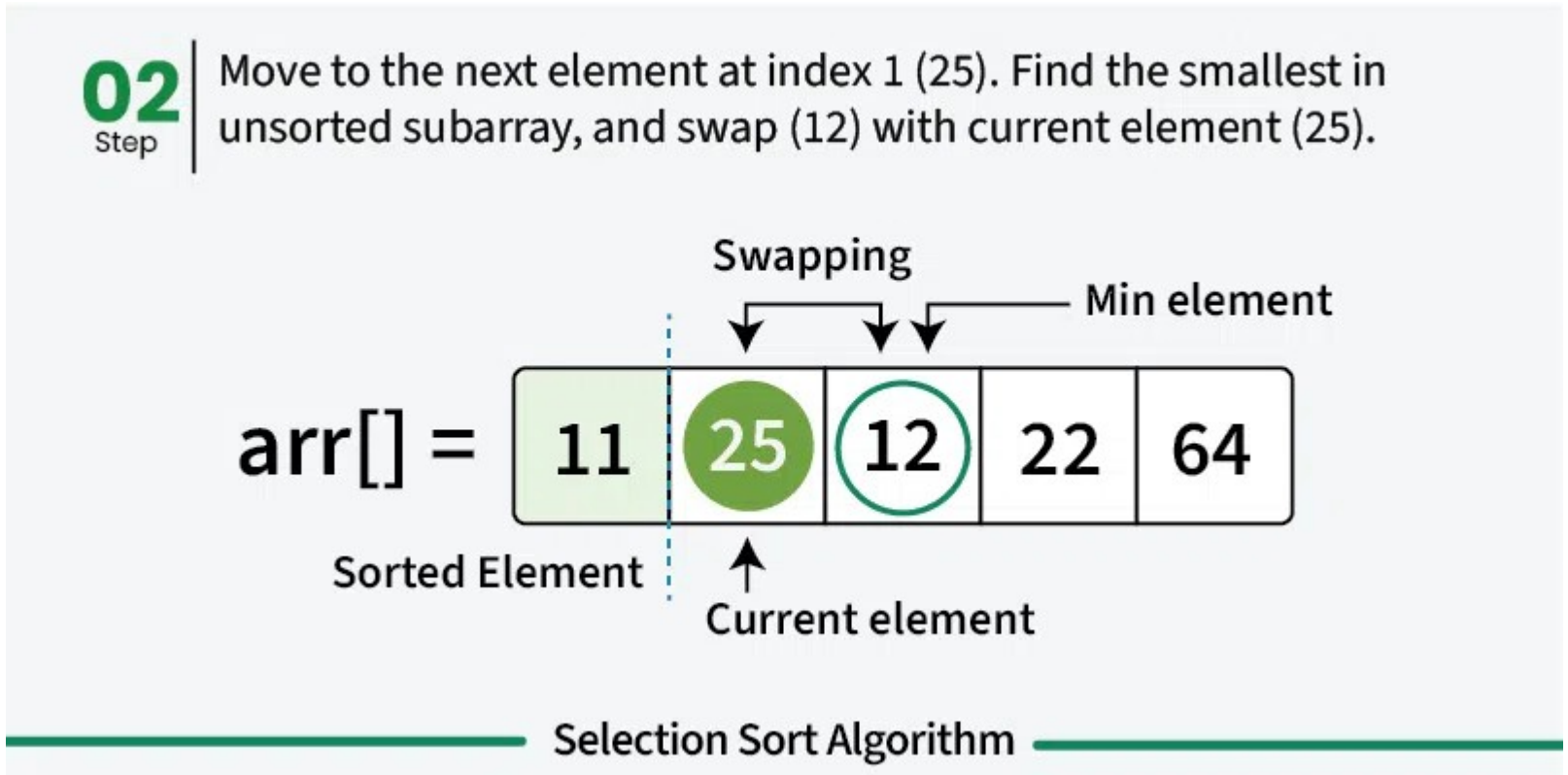
- First, find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
- Then find the smallest *among remaining elements* (or second smallest) and move it to its correct position by swapping.
- Keep doing this until we get all elements moved to correct position.

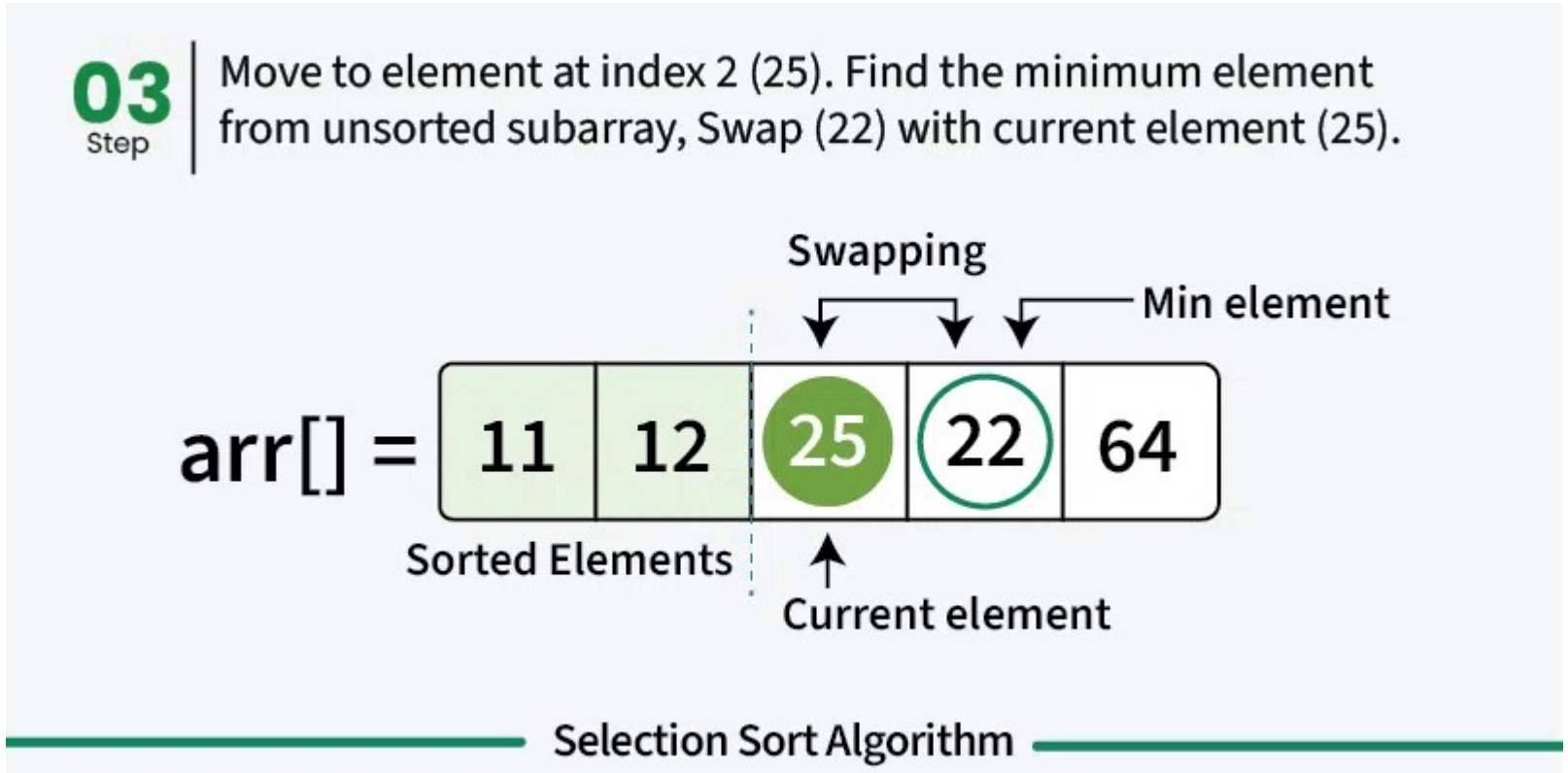
Selection Sort - algorithm

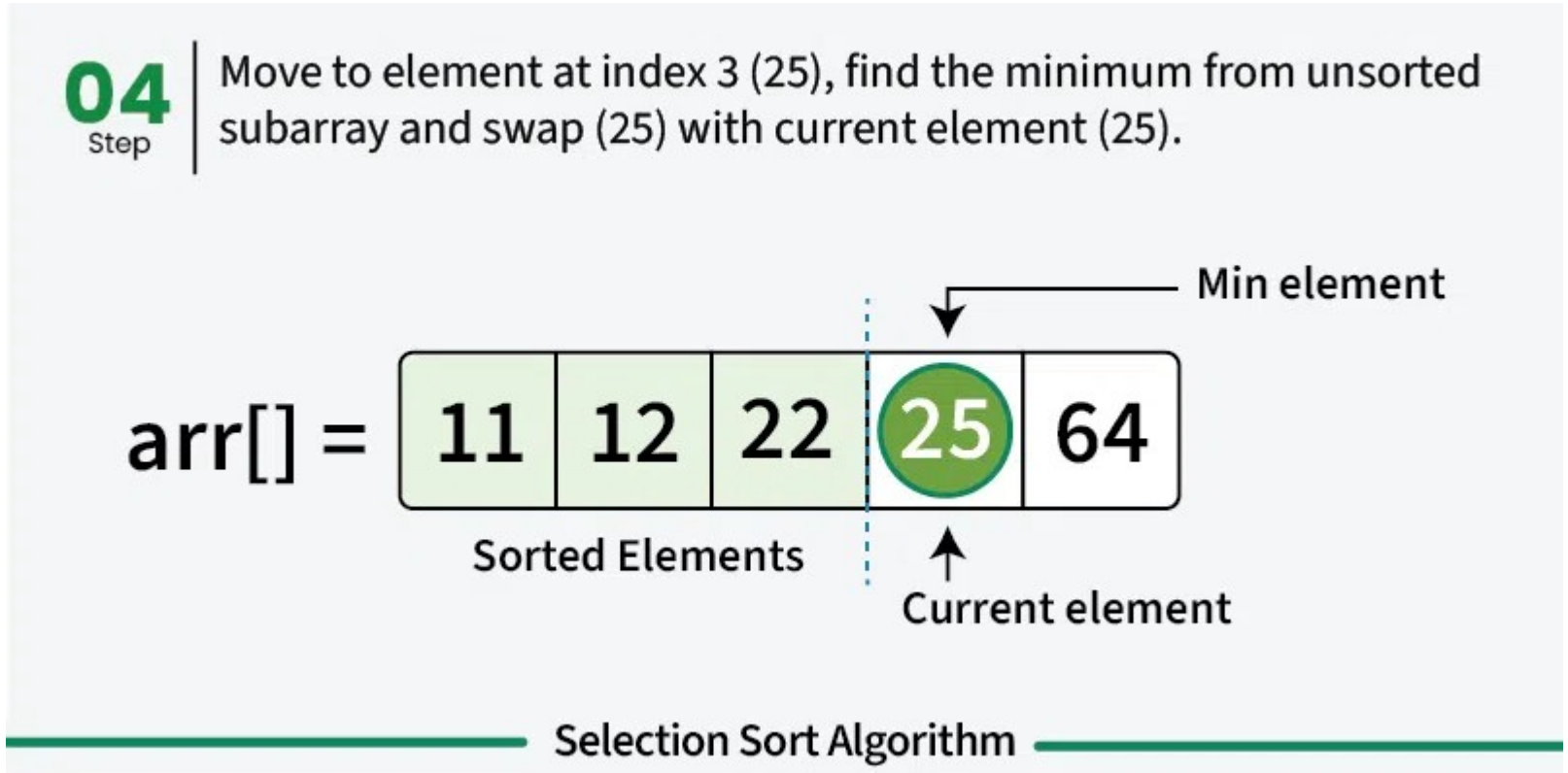
```
def selection_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        # Assume the current position holds
        # the minimum element
        min_idx = i
        # Iterate through the unsorted portion
        # to find the actual minimum
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                # Update min_idx if a smaller element is
found
                min_idx = j
        # Move minimum element to its correct position
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

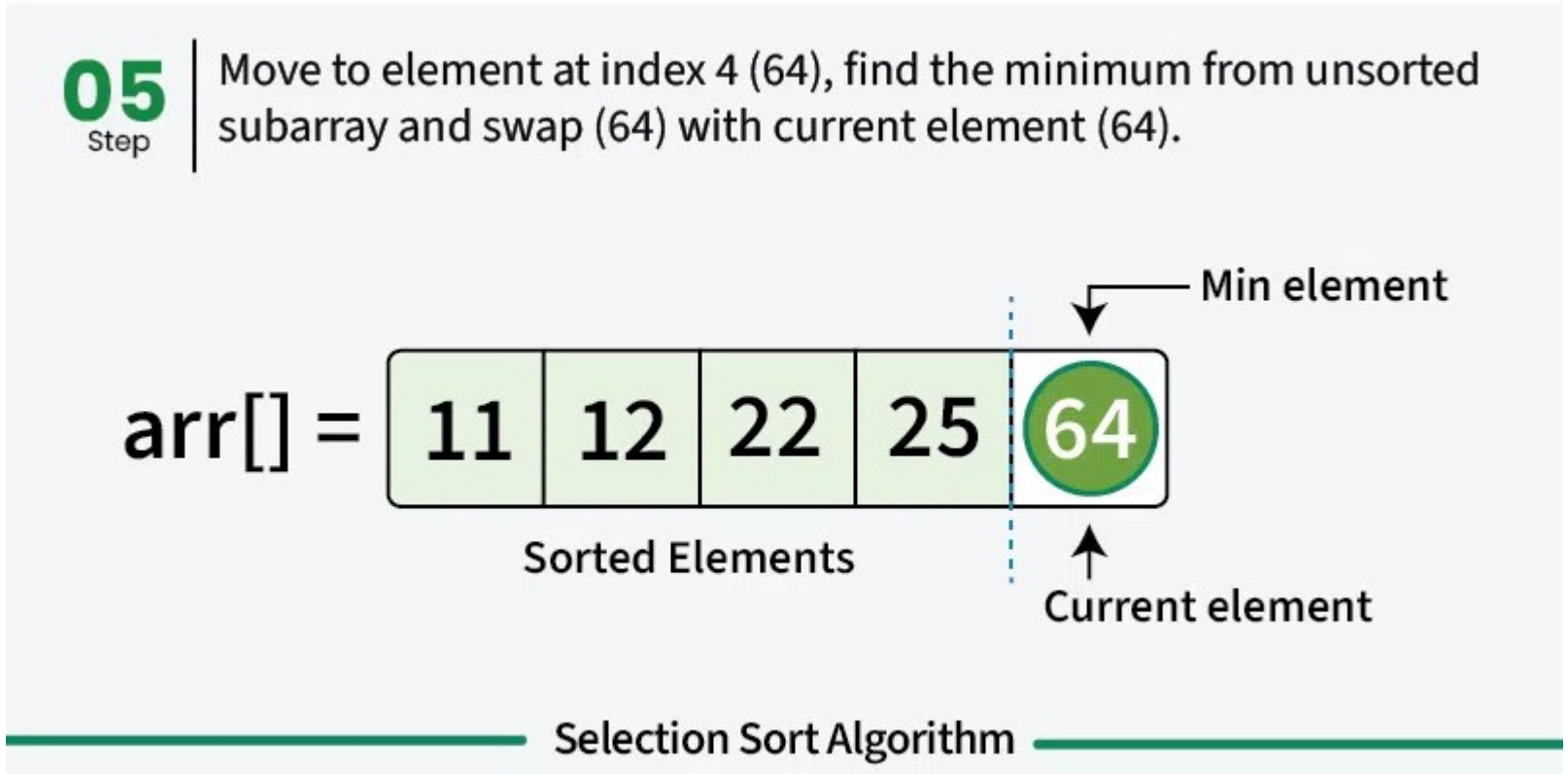
Time Complexity: $O(n^2)$
Auxiliary Space: O(1)

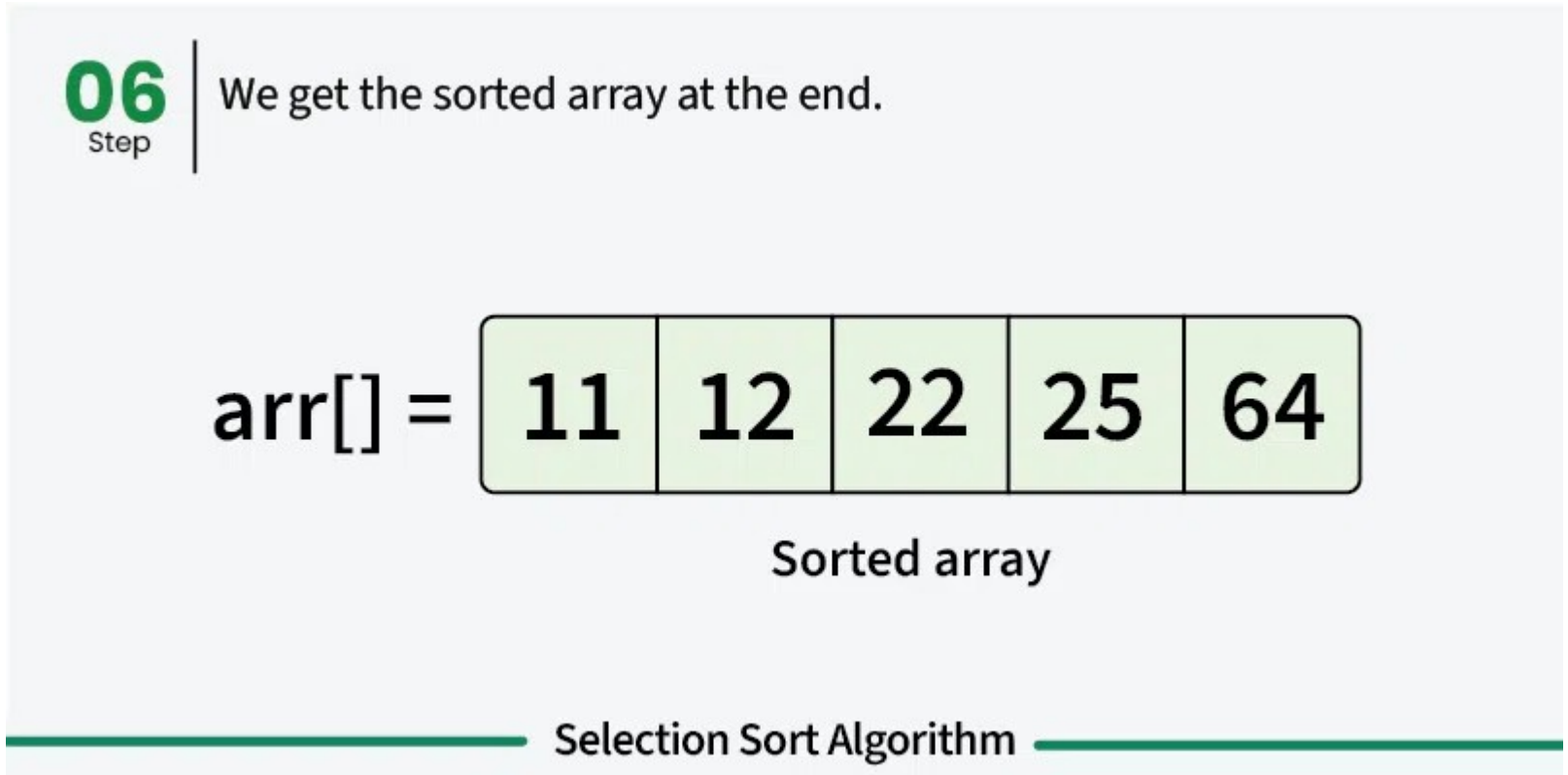

01
Step
Start from the first element at index 0, find the smallest element in the rest of the array which is unsorted, and swap (11) with current element(64).
Swapping Elements
arr[] =
64
25
12
22
11
Current element
Min element
Selection Sort Algorithm


02
Step
Move to the next element at index 1 (25). Find the smallest in unsorted subarray, and swap (12) with current element (25).
Swapping
Min element
arr[] =
11
25
12
22
64
Sorted Element
Current element
Selection Sort Algorithm


03
Step
Move to element at index 2 (25). Find the minimum element from unsorted subarray, Swap (22) with current element (25).
Swapping
Min element
arr[] =
11
12
25
22
64
Sorted Elements
Current element
Selection Sort Algorithm


04
Step
Move to element at index 3 (25), find the minimum from unsorted subarray and swap (25) with current element (25).
Min element
arr[] =
11
12
22
25
64
Sorted Elements
Current element
Selection Sort Algorithm


05
Step
Move to element at index 4 (64), find the minimum from unsorted subarray and swap (64) with current element (64).
Min element
arr[] =
11
12
22
25
64
Sorted Elements
Current element
Selection Sort Algorithm


06
Step
We get the sorted array at the end.
arr[] =
11
12
22
25
64
Sorted array
Selection Sort Algorithm

Selection Sort

GEBZE TECHNICAL UNIVERSITY

Advantages of Selection Sort

- Easy to understand and implement, making it ideal for teaching basic sorting concepts.
- Requires only a constant O(1) extra memory space.
- It requires less number of swaps (or memory writes) compared to many other standard algorithms. Therefore it can be simple algorithm choice when memory writes are costly.

Disadvantages of the Selection Sort

- Selection sort has a time complexity of O(n^2) makes it slower compared to algorithms like Quick Sort or Merge Sort.
- Does not maintain the relative order of equal elements.
- Does not preserve the relative order of items with equal keys which means it is not stable.

Insertion Sort

- Simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list.

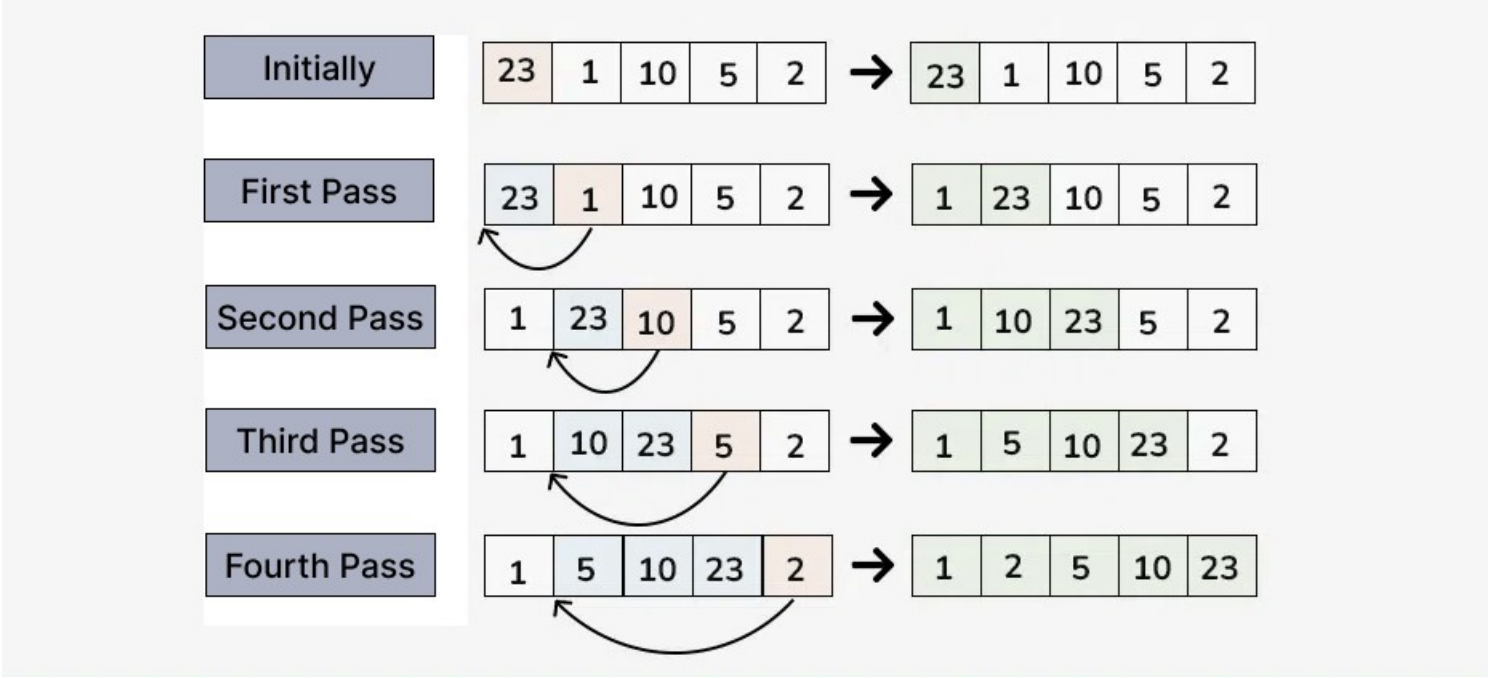
- It is like sorting playing cards in your hands.
- You split the cards into two groups:
  - the sorted cards and the unsorted cards.
  - Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

Insertion Sort – algorithm

```
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

Time Complexity: $O(n^2)$
Auxiliary Space: $O(1)$

Insertion Sort – continued



Insertion Sort – continued

Advantages of Insertion Sort:

- Simple and easy to implement.
- Stable sorting algorithm.
- Efficient for small lists and nearly sorted lists.
- Space-efficient as it is an in-place algorithm.
- Adoptive. the number of inversions is directly proportional to number of swaps. For example, no swapping happens for a sorted array and it takes O(n) time only.

Disadvantages of Insertion Sort:

- Inefficient for large lists.
- Not as efficient as other sorting algorithms (e.g., merge sort, quick sort) for most cases.

Merge Sort

1) Divide: Divide the list or array recursively into two halves until it can no more be divided.

2) Conquer: Each subarray is sorted individually using the merge sort algorithm.

3) Merge: The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

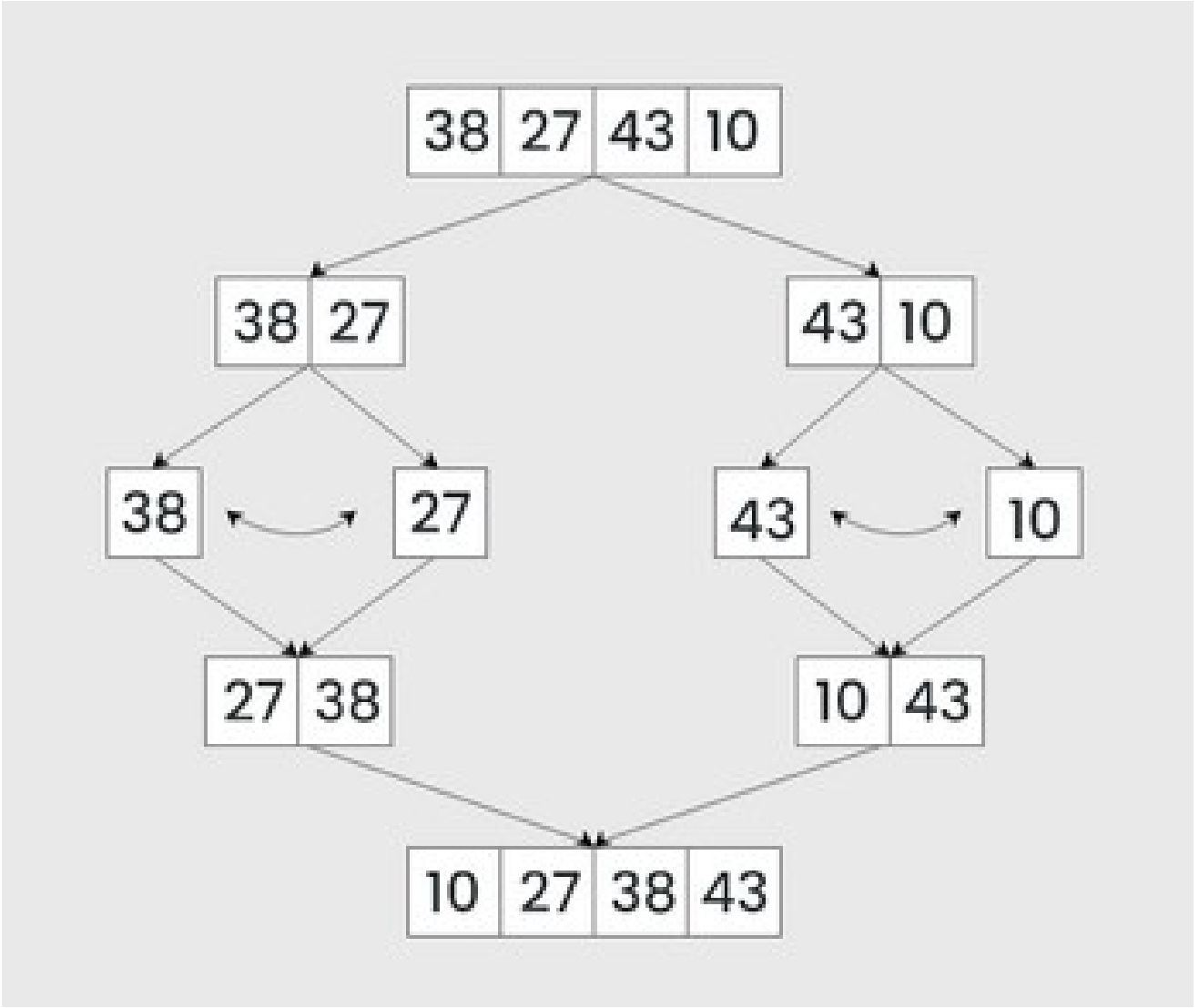
Merge Sort

```
def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]: result.append(left[i]); i += 1
        else: result.append(right[j]); j += 1

    result.extend(left[i:])
    result.extend(right[j:])

    return result
```

```
def mergeSort(arr):
    step = 1  # Starting with sub-arrays of length 1
    length = len(arr)

    while step < length:
        for i in range(0, length, 2 * step):
            left = arr[i:i + step]
            right = arr[i + step:i + 2 * step]
            merged = merge(left, right)
            # Place the merged array back into the original array
            for j, val in enumerate(merged): arr[i + j] = val

        step *= 2  # Double the sub-array length for the next iteration

    return arr
```

```
unsortedArr = [3, 7, 6, -10, 15, 23.5, 55, -13]
sortedArr = mergeSort(unsortedArr)
print("Sorted array:", sortedArr)
```

Merge Sort



Merge Sort

GEBZE TECHNICAL UNIVERSITY

Advantages of Merge Sort:

- Consistent Performance: Merge Sort has a time complexity of O(n log n) in all cases: best, average, and worst. This makes it highly predictable and reliable for sorting large datasets.
- Stable Sorting: Merge Sort is a stable sorting algorithm, meaning it preserves the relative order of equal elements. This is important in applications where the original order of equal elements must be maintained.
- Efficient for Large Datasets: Due to its O(n log n) time complexity, Merge Sort is highly efficient for sorting large datasets. It outperforms simpler algorithms like Bubble Sort, Insertion Sort, and Selection Sort for larger inputs.
- Parallelizable: Merge Sort can be easily parallelized because the divide step splits the array into independent subarrays that can be sorted concurrently. This makes it suitable for multi-threaded or distributed computing environments.
- Suitable for External Sorting: Merge Sort is well-suited for external sorting, where the data to be sorted is too large to fit into memory. It efficiently handles data stored on disk by dividing it into smaller chunks, sorting them, and then merging them.

Merge Sort

GEBZE TECHNICAL UNIVERSITY

Disadvantages of Merge Sort:

- Space Complexity: Merge Sort requires O(n) additional space for the temporary array used during merging. This makes it less space-efficient compared to in-place sorting algorithms like Quick Sort, Heap Sort, or Bubble Sort.
- Not In-Place: Merge Sort is not an in-place sorting algorithm, meaning it requires additional memory proportional to the size of the input array. This can be a limitation in memory-constrained environments.
- Slower for Small Datasets: For small datasets, Merge Sort may be slower than simpler algorithms like Insertion Sort or Bubble Sort due to its higher overhead (e.g., recursive calls and merging).
- Recursive Overhead: Merge Sort relies heavily on recursion, which can lead to additional overhead in terms of function calls and stack space. In some cases, this can impact performance, especially for very large datasets.
- Complex Implementation: While the concept of Merge Sort is straightforward, its implementation can be more complex compared to simpler algorithms like Bubble Sort or Insertion Sort. This can make it harder to debug and maintain.