**DATA Structure**

1. **Arrays**

An array is a structured, fixed-size collection of elements stored in a specific order, allowing quick access by index.

* **Advantages**: Arrays provide fast access to elements through indexing. For example, in a list of days [Sunday, Monday, Tuesday], accessing Tuesday by index (2) is immediate.
* **Drawbacks**: Arrays are inefficient for frequent insertions or deletions. If you want to insert an element at the beginning of [1, 2, 3], all elements must shift, which is slow. Arrays are also fixed in size, so they can’t grow or shrink dynamically.
* **When to Use**: Use arrays for static datasets where the number of items is fixed, such as storing months of the year.
* **When Not to Use**: Avoid arrays if data is highly dynamic, like online users that frequently join or leave a session.

1. **Linear Search in Arrays**

Linear Search is the simplest searching algorithm where we search for a **target element** by sequentially checking each element in a list or array until we find the desired element or reach the end of the list. It operates in a **step-by-step manner**, making it suitable for small datasets.

**How Linear Search Works**

1. Start at the first element of the list.
2. Compare the current element with the target element.
3. If they match, return the index of the element.
4. If not, move to the next element and repeat.
5. If the end of the list is reached and the target is not found, return -1.

* **Advantages:** Works on unsorted arrays; simple and easy to implement. For instance, in an unsorted list of names, [Anna, Mark, Zara], linear search can still locate “Mark” by checking sequentially.
* **Drawbacks**: Inefficient for large arrays. Each element must be checked individually, so in a list of 10,000 items, finding the target may require 10,000 checks.
* **When to Use**: Use linear search for small or unsorted arrays where simplicity is preferred, such as checking attendance in a short list.
* **When Not to Use**: Avoid for large, sorted datasets where binary search would be much faster.

1. **Binary Search in Arrays**

**Binary Search** is an efficient algorithm used to find the position of a target element in a **sorted array**. It works by dividing the dataset into halves during each step and narrowing down the search range. This significantly reduces the number of comparisons needed to locate the target element compared to a linear search.

**How Does Binary Search Work?**

Binary search follows these steps:

1. **Initial Setup:**
   * Define the **low** (start of the array) and **high** (end of the array) indices.
   * Compute the **midpoint** of the current range using the formula: mid=low+(high−low)/2\text{mid} = \text{low} + (\text{high} - \text{low}) / 2mid=low+(high−low)/2
2. **Compare Midpoint with Target:**
   * Check if the element at the mid index matches the target:
     + If **yes**, return the index of the element.
     + If **no**, adjust the search range:
       - If the target is **less than** the element at mid, search the **left half** by updating high = mid - 1.
       - If the target is **greater than** the element at mid, search the **right half** by updating low = mid + 1.
3. **Repeat Steps 1 & 2:**
   * Continue halving the search range until the target is found or the range becomes invalid (i.e., low > high).
4. **Target Not Found:**
   * If the search range becomes invalid, return -1.

* **Advantages**: Highly efficient for large, sorted datasets. In a sorted list of book titles, binary search can locate “Zebra” in far fewer steps by halving the array each time, making it fast for large collections.
* **Drawbacks**: Only works on sorted arrays, so if the array isn’t sorted, it won’t work. Additionally, if the dataset is dynamic and requires frequent insertions or deletions, it can become inefficient to maintain the sorted order.
* **When to Use**: Ideal for large, sorted arrays where fast searching is needed, like in a dictionary lookup.
* **When Not to Use**: Avoid binary search for unsorted or frequently changing datasets, as sorting the array each time would be inefficient.

1. **Big O Notation**

**Big O Notation** is used to describe the **time complexity** of an algorithm, or how efficiently an algorithm scales with the size of the input (denoted as nnn). It gives an upper bound on the algorithm's growth rate as the input size increases. In Big O, we focus on the dominant term and ignore constants to simplify the comparison between algorithms.

**Common Big O Complexity Classes:**

* **O(1)**: Constant time – the runtime is unaffected by input size.
* **O(log n)**: Logarithmic time – the runtime grows slowly as input size increases.
* **O(n)**: Linear time – the runtime grows proportionally with input size.
* **O(n log n)**: Linearithmic time – common in efficient sorting algorithms.
* **O(n^2)**: Quadratic time – grows quickly; often found in algorithms with nested loops.

**Linear Search – O(n) Complexity**

**Linear Search** has a time complexity of **O(n)** because, in the worst case, it has to check every element in the array.

* **Example**: If searching for the last element in an unsorted list of names, the search goes through each name until it reaches the target.
* **Identifying O(n)**: The number of comparisons scales directly with the number of elements; if there are 10 elements, it may take 10 checks.

**Graph:**

The graph of an O(n) complexity is a **straight line** starting at the origin, with runtime (y-axis) increasing proportionally with input size (x-axis).

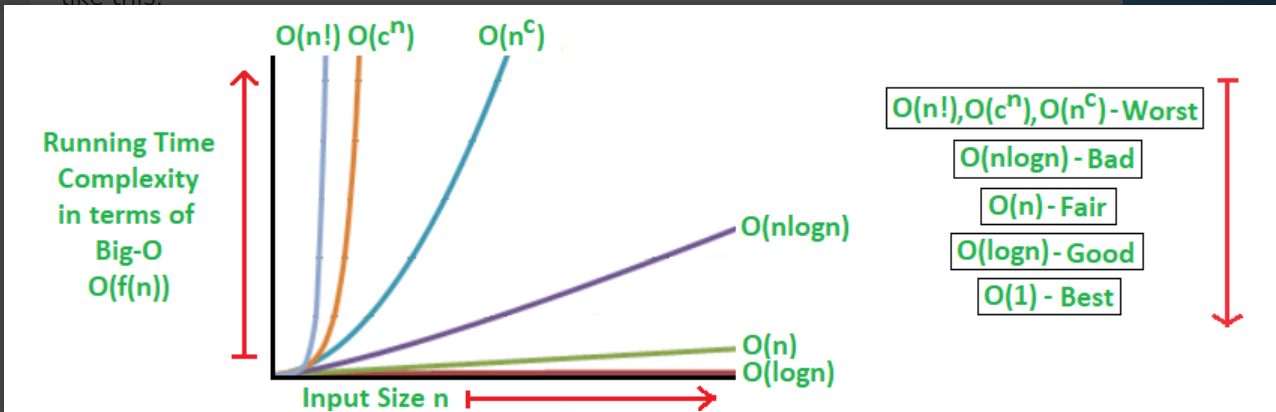
**Binary Search – O(log n) Complexity**

**Binary Search** has a time complexity of **O(log n)** because it repeatedly divides the array in half, reducing the search space exponentially.

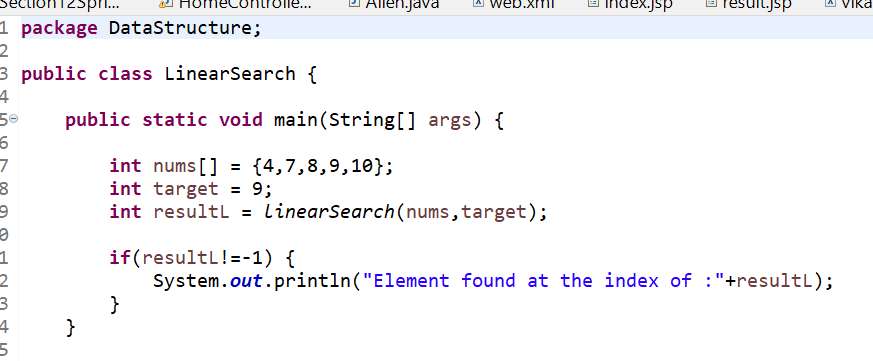
* **Example**: In a sorted list of book titles, each comparison reduces the search space by half until the target is found, minimizing the number of checks.
* **Identifying O(log n)**: Since binary search eliminates half of the elements each step, the number of steps grows logarithmically with input size.

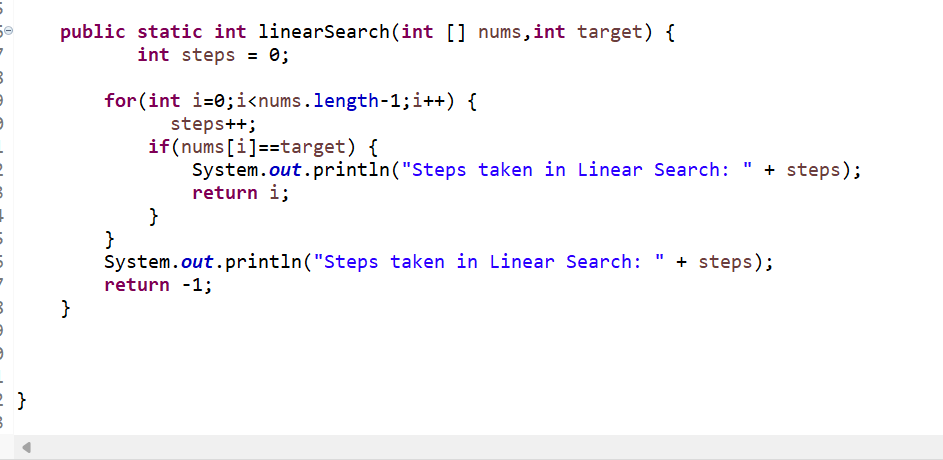
**Graph:**

The graph of an O(log n) complexity shows a **curve that flattens** as input size increases, indicating slower growth in runtime.

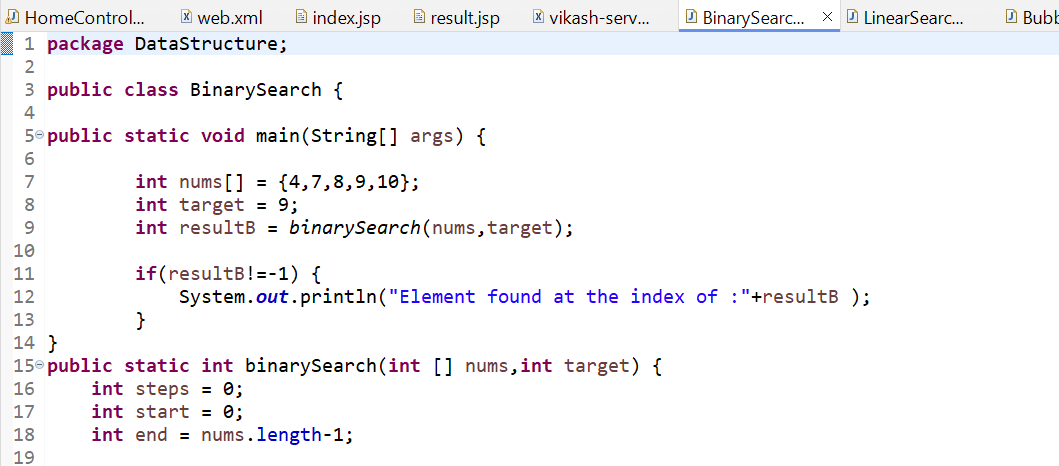


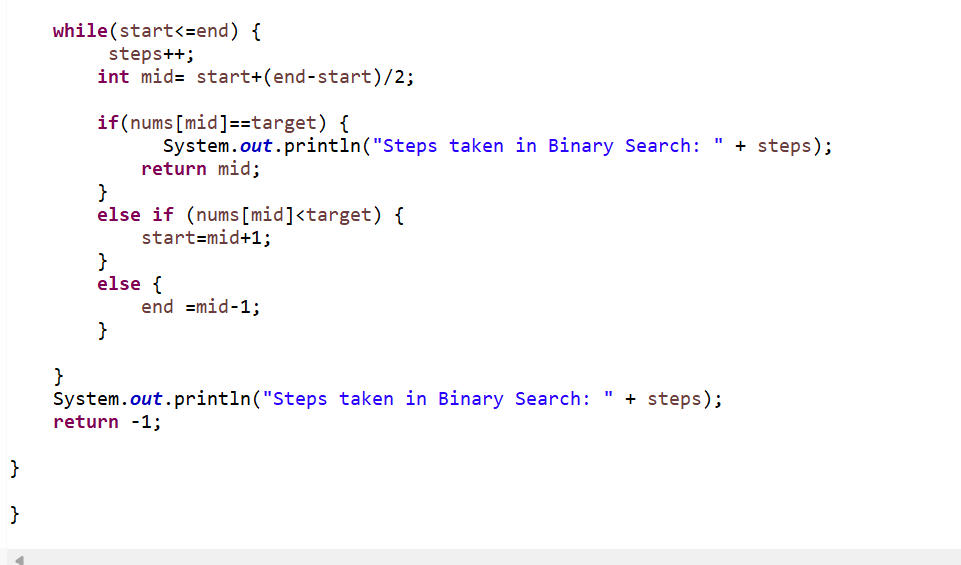
**Linear Search Example**

****

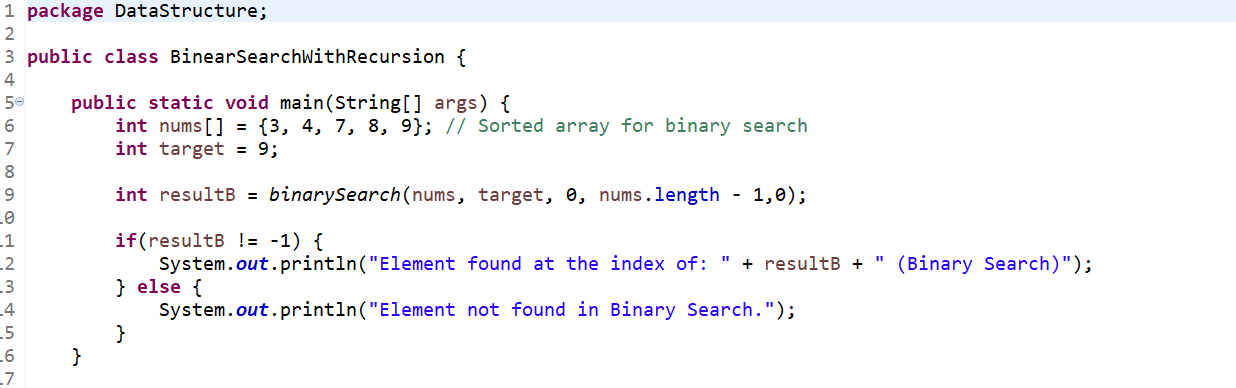
****

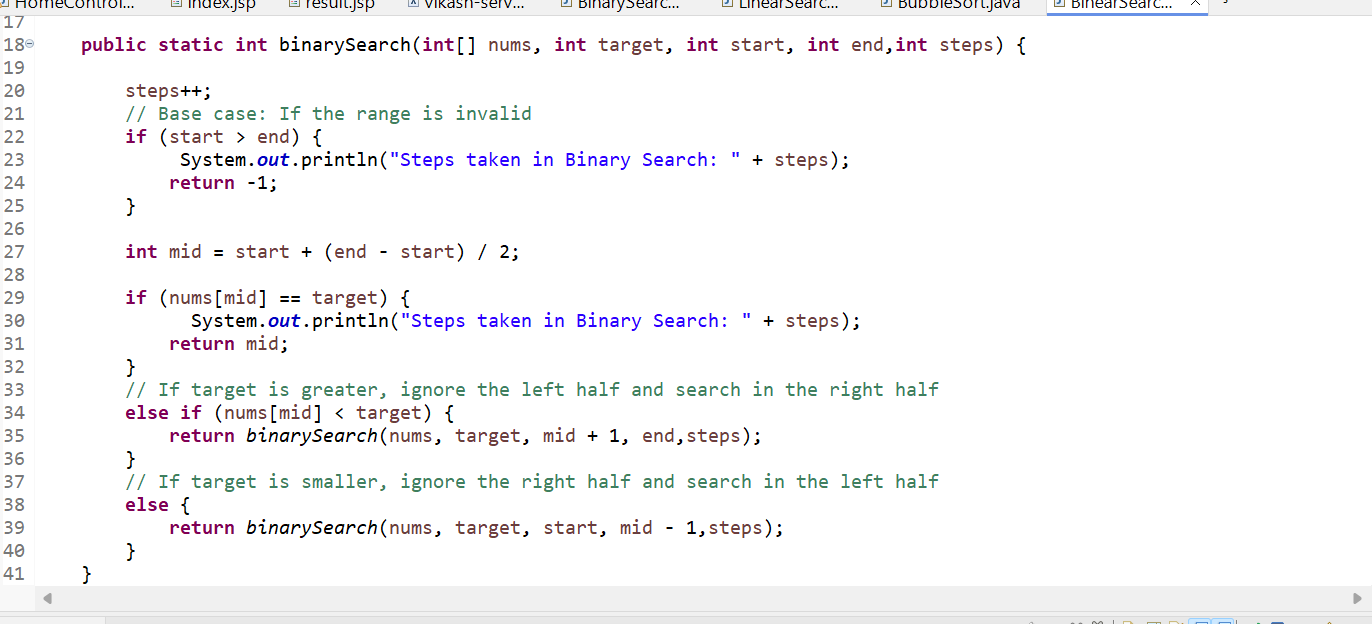
1. **Binary search Example**

****

****

1. **Binary search Example with recursion**

****

****

**9. Bubble Sort**

**Bubble Sort** is a simple sorting algorithm that repeatedly steps through a list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is sorted. The algorithm gets its name because smaller elements "bubble" to the top (start of the array), and larger elements sink to the bottom (end of the array) during each pass.

**How Does Bubble Sort Work?**

The algorithm operates by repeatedly iterating over the list and performing the following steps:

1. **Compare Adjacent Elements:**
   * For each pair of adjacent elements, compare them.
2. **Swap if Necessary:**
   * If the first element is greater than the second, swap them to move the larger element towards the end of the list.
3. **Repeat Until Sorted:**
   * Perform multiple passes over the list.
   * After each pass, the largest unsorted element is guaranteed to be in its correct position (at the end).
4. **Optimize Early Exit (Optional):**
   * If no swaps are made during a pass, the list is already sorted, and the algorithm can terminate early.

**Initial Array:**

For example, consider the array:

csharp

Copy code

[5, 3, 8, 4, 2]

**Pass 1:**

* Compare 5 and 3: Swap → [3, 5, 8, 4, 2]
* Compare 5 and 8: No Swap → [3, 5, 8, 4, 2]
* Compare 8 and 4: Swap → [3, 5, 4, 8, 2]
* Compare 8 and 2: Swap → [3, 5, 4, 2, 8]

**Result after Pass 1:** [3, 5, 4, 2, 8] (Largest element, 8, is now at the end).

**Pass 2:**

* Compare 3 and 5: No Swap → [3, 5, 4, 2, 8]
* Compare 5 and 4: Swap → [3, 4, 5, 2, 8]
* Compare 5 and 2: Swap → [3, 4, 2, 5, 8]

**Result after Pass 2:** [3, 4, 2, 5, 8] (Second largest element, 5, is now in place).

**Pass 3:**

* Compare 3 and 4: No Swap → [3, 4, 2, 5, 8]
* Compare 4 and 2: Swap → [3, 2, 4, 5, 8]

**Result after Pass 3:** [3, 2, 4, 5, 8] (Third largest element, 4, is now in place).

**Pass 4:**

* Compare 3 and 2: Swap → [2, 3, 4, 5, 8]

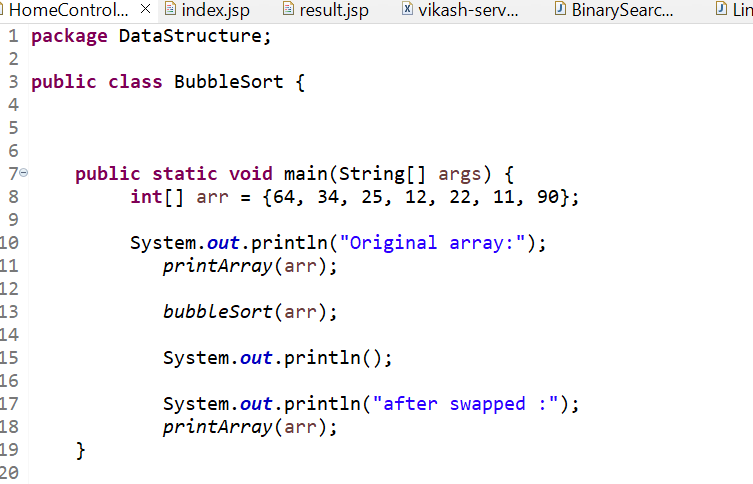
**Result after Pass 4:** [2, 3, 4, 5, 8] (List is now fully sorted).

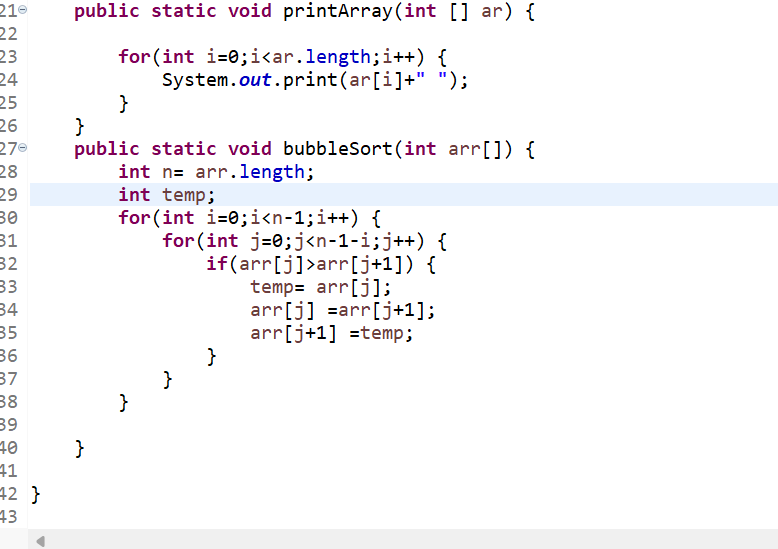
**Pass 5:**

* No swaps are made, so the algorithm terminates.

**Characteristics of Bubble Sort**

1. **Time Complexity:**
   * **Best Case (Already Sorted):** O(n)O(n)O(n) (With early exit optimization).
   * **Worst Case (Reversed List):** O(n2)O(n^2)O(n2).
   * **Average Case:** O(n2)O(n^2)O(n2).
2. **Space Complexity:**
   * O(1)O(1)O(1) (In-place sorting).

****

****

**Dry Run**

**Pass 1 (i = 0):**

Array before Pass 1: {64, 34, 25, 12, 22, 11, 90}

1. Compare 64 and 34: Swap → {34, 64, 25, 12, 22, 11, 90}
2. Compare 64 and 25: Swap → {34, 25, 64, 12, 22, 11, 90}
3. Compare 64 and 12: Swap → {34, 25, 12, 64, 22, 11, 90}
4. Compare 64 and 22: Swap → {34, 25, 12, 22, 64, 11, 90}
5. Compare 64 and 11: Swap → {34, 25, 12, 22, 11, 64, 90}
6. Compare 64 and 90: No Swap → {34, 25, 12, 22, 11, 64, 90}

**Result after Pass 1:** {34, 25, 12, 22, 11, 64, 90}

**Pass 2 (i = 1):**

Array before Pass 2: {34, 25, 12, 22, 11, 64, 90}

1. Compare 34 and 25: Swap → {25, 34, 12, 22, 11, 64, 90}
2. Compare 34 and 12: Swap → {25, 12, 34, 22, 11, 64, 90}
3. Compare 34 and 22: Swap → {25, 12, 22, 34, 11, 64, 90}
4. Compare 34 and 11: Swap → {25, 12, 22, 11, 34, 64, 90}

**Result after Pass 2:** {25, 12, 22, 11, 34, 64, 90}

**Pass 3 (i = 2):**

Array before Pass 3: {25, 12, 22, 11, 34, 64, 90}

1. Compare 25 and 12: Swap → {12, 25, 22, 11, 34, 64, 90}
2. Compare 25 and 22: Swap → {12, 22, 25, 11, 34, 64, 90}
3. Compare 25 and 11: Swap → {12, 22, 11, 25, 34, 64, 90}

**Result after Pass 3:** {12, 22, 11, 25, 34, 64, 90}

**Pass 4 (i = 3):**

Array before Pass 4: {12, 22, 11, 25, 34, 64, 90}

1. Compare 12 and 22: No Swap → {12, 22, 11, 25, 34, 64, 90}
2. Compare 22 and 11: Swap → {12, 11, 22, 25, 34, 64, 90}

**Result after Pass 4:** {12, 11, 22, 25, 34, 64, 90}

**Pass 5 (i = 4):**

Array before Pass 5: {12, 11, 22, 25, 34, 64, 90}

1. Compare 12 and 11: Swap → {11, 12, 22, 25, 34, 64, 90}

**Result after Pass 5:** {11, 12, 22, 25, 34, 64, 90}

**Pass 6 (i = 5):**

Array before Pass 6: {11, 12, 22, 25, 34, 64, 90}

* No comparisons are needed as the array is already sorted.