**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 checks each element in sequence to find the target. It doesn’t require sorting, making it useful across various array types.

* **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 efficiently finds a target in a sorted array by halving the search space with each comparison.

* **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).

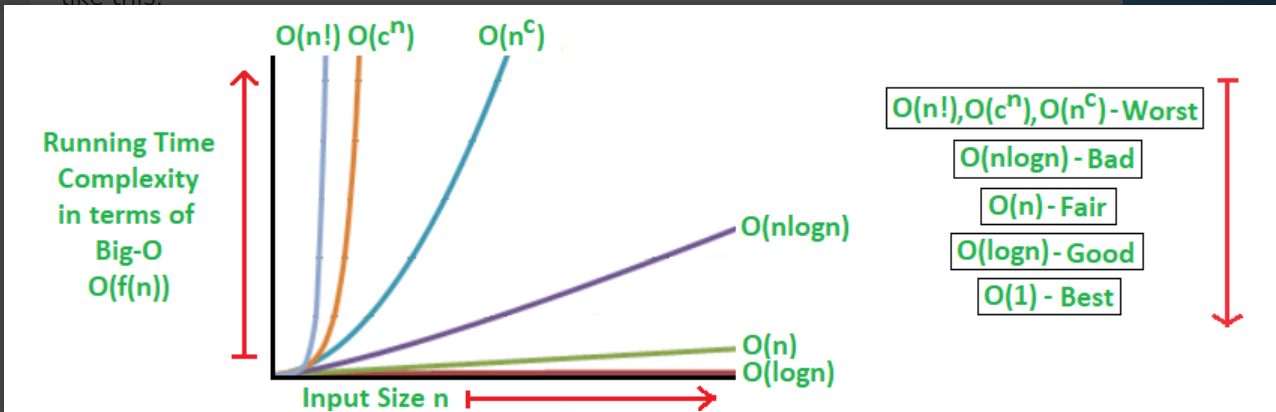
**Binary Search – O(log n) Complexity**

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

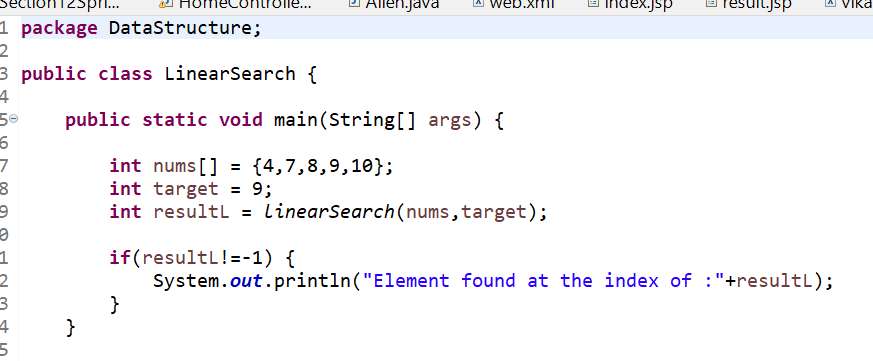
* **Example**: In a sorted list of book titles, each comparison reduces the search space by half until the target is found, minimizing the number of checks.
* **Identifying O(log n)**: Since binary search eliminates half of the elements each step, the number of steps grows logarithmically with input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

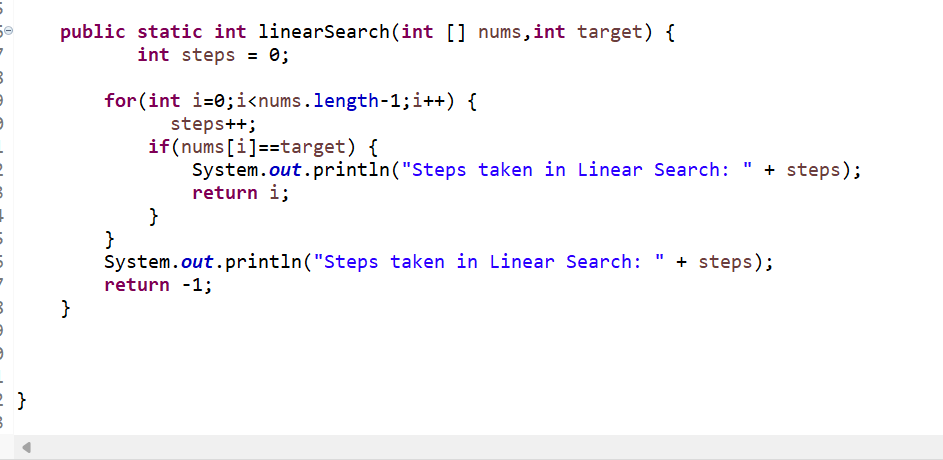
**Graph:**

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

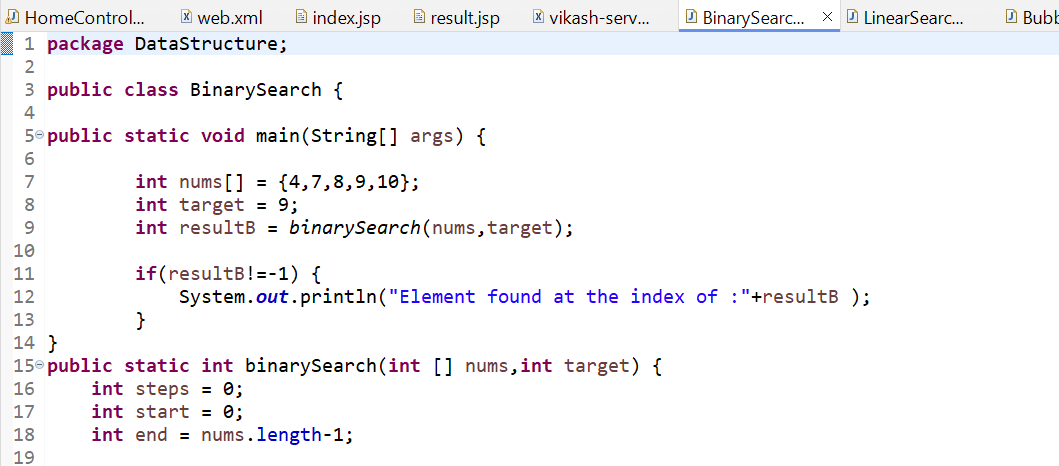


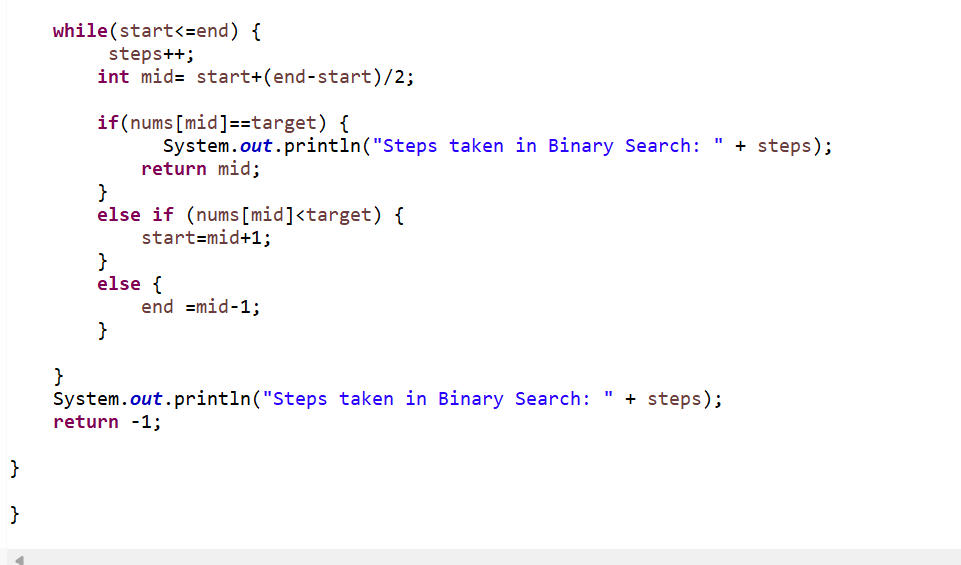
**Linear Search Example**

****

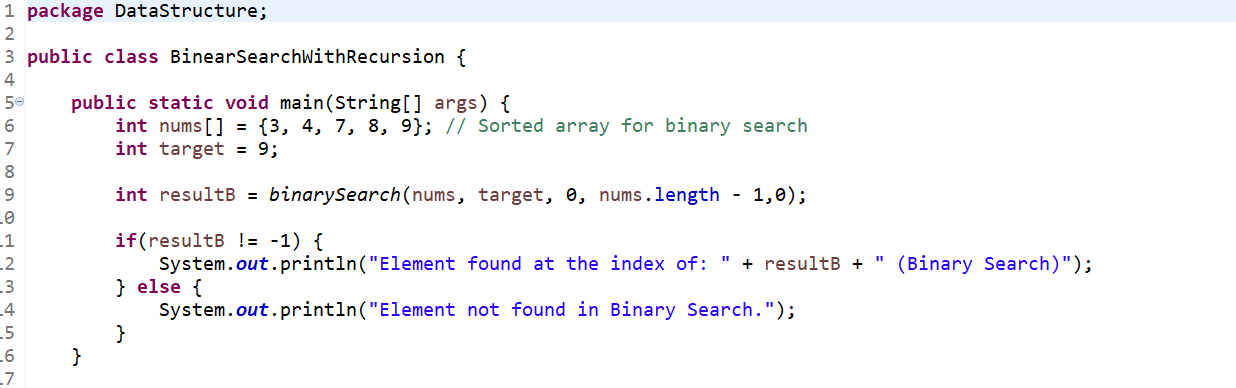
****

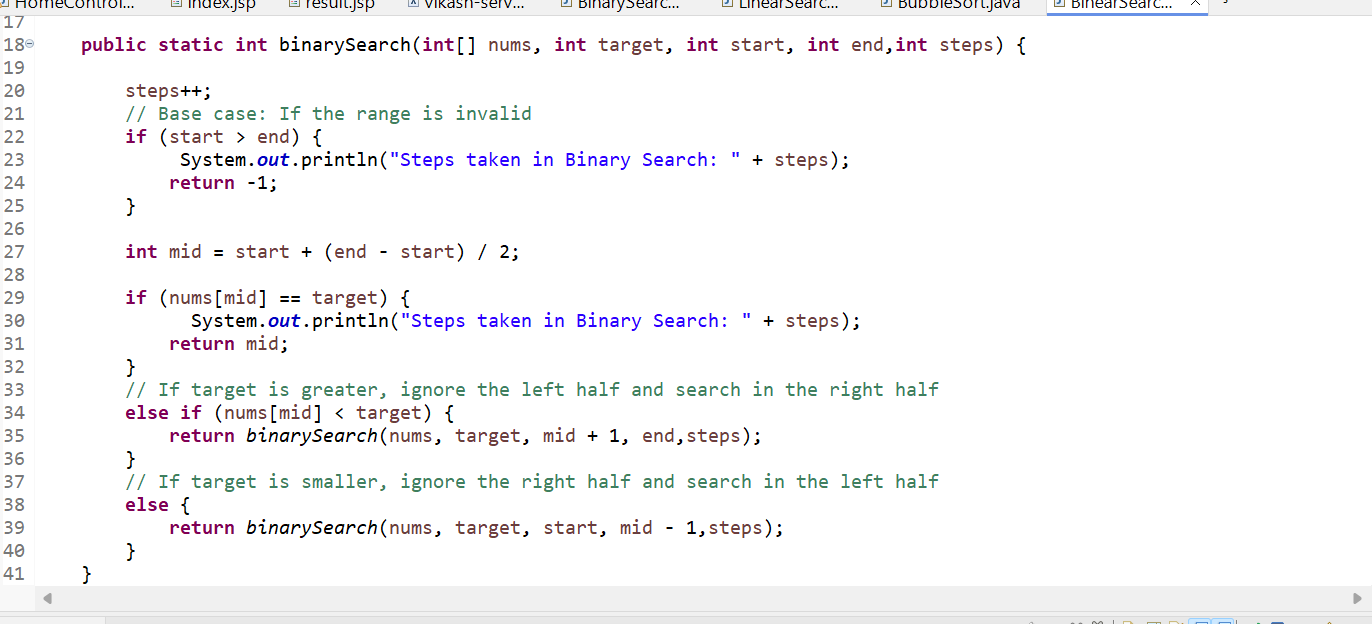
1. **Binary search Example**

****

****

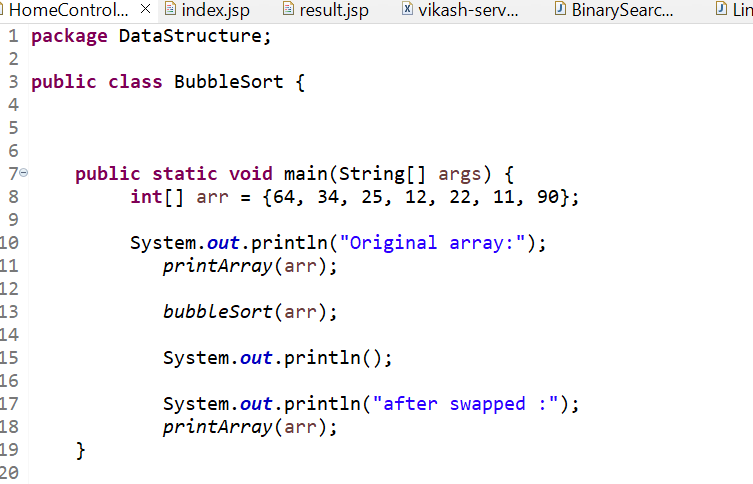
1. **Binary search Example with recurssion**

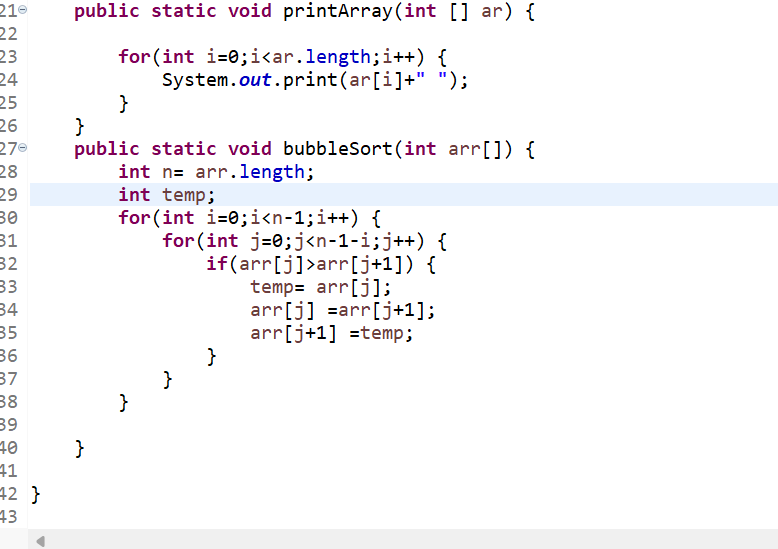
****

****

**9. Bubble Sort**

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted.

****

****