Step 1: Understand Asymptotic Notation

What is Big O Notation?

Big O notation describes how the runtime of an algorithm increases with input size. It helps estimate scalability and performance.

- **O(1)**: Constant time
- **O(log n)**: Logarithmic time (e.g., Binary Search)
- **O(n)**: Linear time (e.g., Linear Search)
- **O(n log n)**: Linearithmic (e.g., Merge Sort)
- **O(n²)**: Quadratic (e.g., Nested loops)

Best, Average, and Worst Case in Search

Algorithm	Best Case	Average Case	Worst Case
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

- **Linear Search**: Checks every element until a match is found.
- **Binary Search**: Works on sorted arrays, repeatedly divides the search space.

Step 2: Setup - Define the `Product` Class

File: Product.java

```
public class Product {

    private int productId;
    private String productName;
    private String category;

    public Product(int productId, String productName, String category) {
        this.productId = productId;
        this.productName = productName;
        this.category = category;
    }

    public int getProductId() {
        return productId;
    }

    public String getProductName() {
        return productName;
    }
```

```java
    public String getCategory() {
        return category;
    }
}
```

Step 3: Implementation of Search Algorithms

File: SearchUtility.java

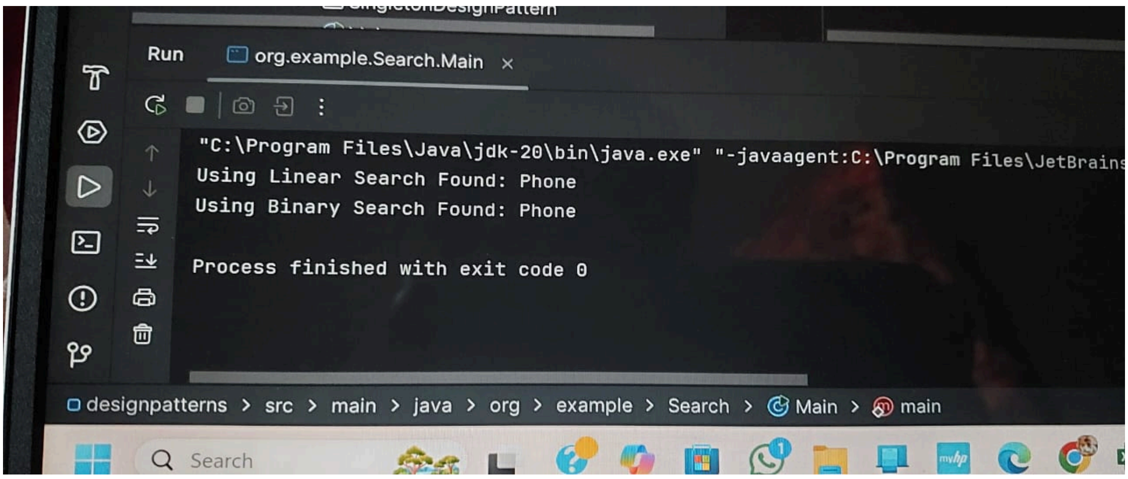
```java
public class SearchUtility {

    // Linear Search by productId
    public static Product linearSearch(Product[] products, int targetId) {
        for (Product product : products) {
            if (product.getProductId() == targetId) {
                return product;
            }
        }
        return null;
    }

    // Binary Search (sorted by productId)
    public static Product binarySearch(Product[] products, int targetId) {
        int low = 0;
        int high = products.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;
            int midId = products[mid].getProductId();

            if (midId == targetId) {
                return products[mid];
            } else if (midId < targetId) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return null;
    }
}
```

Test the Search Functionality

File: Main.java

```
import java.util.Arrays;
import java.util.Comparator;

public class Main {
    public static void main(String[] args) {
        Product[] products = {
            new Product(101, "Laptop", "Electronics"),
            new Product(203, "Shoes", "Fashion"),
            new Product(150, "Phone", "Electronics"),
            new Product(99, "T-shirt", "Fashion")
        };

        // Linear Search (unsorted)
        Product found1 = SearchUtility.linearSearch(products, 150);
        System.out.println(found1 != null ? "Found: " + found1.getProductName() :
"Product not found");

        // Sort for Binary Search
        Arrays.sort(products, Comparator.comparingInt(Product::getProductId));

        // Binary Search
        Product found2 = SearchUtility.binarySearch(products, 150);
        System.out.println(found2 != null ? "Found: " + found2.getProductName() :
"Product not found");
    }
}
```

Output:



Step 4: Analysis

Search Method	Time Complexity	Requires Sorting	Suitable For
Linear Search	O(n)	No	Small or unsorted data
Binary Search	O(log n)	Yes	Large sorted datasets