**Big O Notation** is a mathematical way to describe **how the runtime (or space) of an algorithm grows** as the size of the input increases.

It helps us **compare algorithms** in terms of their **efficiency**, regardless of hardware or programming language.

**Best, Average, and Worst Case in Search Operations**

These refer to **different performance scenarios** depending on the input:

**1. Best Case**

* The ideal, fastest scenario.
* **Example (Linear Search)**: The item is at the **first position**.
* Time Complexity: **O(1)**

**2. Average Case**

* Expected scenario if all positions are equally likely.
* **Example**: The item is somewhere in the middle.
* Time Complexity: **O(n/2)** → Simplified as **O(n)**

**3. Worst Case**

* The slowest scenario.
* **Example**: The item is **not present** or at the last position.
* Time Complexity: **O(n)** for linear search, **O(log n)** for binary search.

Code:

package DSA;

class Product {

int productId;

String productName;

String category;

public Product(int productId, String productName, String category) {

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

}

class LinearSearch{

        public int search(Product[] p,int productId){

            for(int i=0;i<p.length;i++){

                if(p[i].productId == productId){

                    return i;

                }

            }

            return -1;

        }

    }

class BinarySearch{

        public int search(Product[] p,int productId){

            int low = 0;

            int high = p.length - 1;

            while (low <= high) {

                int mid = (low + high) / 2;

                if (p[mid].productId == productId) {

                    return mid;

                } else if (p[mid].productId < productId) {

                    low = mid + 1;

                } else {

                    high = mid - 1;

                }

            }

            return -1;

        }

}

public class Main {

    public static void main(String[] args) {

        Product[] products = {

            new Product(101, "Laptop", "Electronics"),

            new Product(205, "Shirt", "Apparel"),

            new Product(150, "Book", "Stationery"),

            new Product(120, "Phone", "Electronics"),

            new Product(180, "Shoes", "Footwear")

        };

        // For binary search, create a sorted array by productId

        Product[] sortedProducts = products.clone();

        java.util.Arrays.sort(sortedProducts, (a, b) -> Integer.compare(a.productId, b.productId));

        // Example usage

        LinearSearch linearSearch = new LinearSearch();

        int linearIndex = linearSearch.search(products, 150);

        BinarySearch binarySearch = new BinarySearch();

        int binaryIndex = binarySearch.search(sortedProducts, 150);

        System.out.println("Linear Search Index: " + linearIndex);

        System.out.println("Binary Search Index: " + binaryIndex);

        if (linearIndex != -1) {

            System.out.println("Found using Linear Search: " + products[linearIndex].productName);

        } else {

            System.out.println("Product not found using Linear Search.");

        }

        if (binaryIndex != -1) {

            System.out.println("Found using Binary Search: " + sortedProducts[binaryIndex].productName);

        } else {

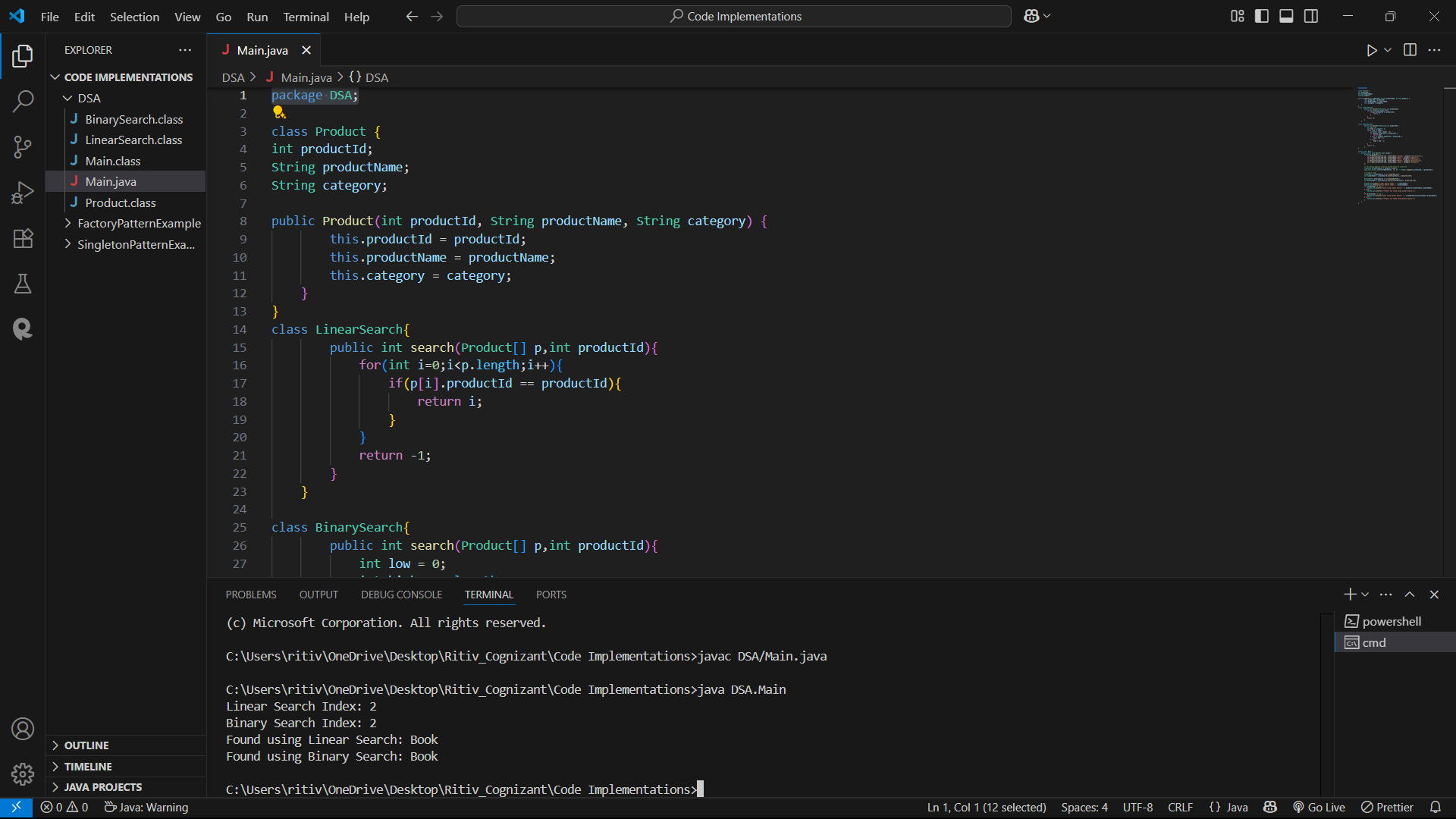
            System.out.println("Product not found using Binary Search.");

        }

    }

}

Output:



**🔹 Time Complexity Comparison: Linear Search vs Binary Search**

| **Feature** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Time Complexity (Best)** | O(1) – item found at start | O(1) – item is the middle element |
| **Time Complexity (Average)** | O(n) | O(log n) |
| **Time Complexity (Worst)** | O(n) – item not found or at end | O(log n) – keep dividing till one element |
| **Data Requirement** | Works on **unsorted** data | Requires **sorted** data |
| **Implementation Simplicity** | Very simple | Slightly more complex (uses divide & conquer) |

**🔹 Which Algorithm is More Suitable?**

That depends on your **platform’s data characteristics**:

**✅ Use Linear Search if:**

* Data is **unsorted** or **small** in size
* You cannot afford the overhead of sorting
* Simplicity matters more than performance

**✅ Use Binary Search if:**

* Data is already **sorted**
* You need **faster search** on **large datasets**
* You care about **scalability and performance**