**E-commerce Platform**

**Cognizant Deepskilling Assessment**

By Vaishnavi K

20/6/2025

Understanding Asymptotic Notation

**Big O Notation**: It describes the worst-case running time of an algorithm. It tells us how long an algorithm will take to run when the amount of data (input) grows.

For example,

if you have more products to search through, Big O notation describes how quickly the search time increases.

If an array has 1000 elements, and you have to potentially check every element, that is **O(n)** — it grows linearly with the number of elements.

**Big Omega Notation**: It describes the best-case running time of an algorithm.

For example,

If you search for an item and it is found in the first position, that takes only one step — this is **Omega(1)**.

**Big Theta Notation**: It describes the average-case running time of an algorithm, when best and worst cases are roughly the same.

For example,

If an algorithm takes roughly the same time regardless of input, it's

**theta(n)**, meaning it's both the best and worst behavior.

When we search for a product, it can be found in different scenarios like best, average and worst cases.

**Best Case :**

The best possible situation. The item we are looking for is found immediately.

Example: We search for a product and it’s the very first one we check.

**O(1)** (just one step)

**Average Case** :

The item we’re looking for is somewhere in the middle. We have to look through roughly half the data.

Example: We have 100 products, and we find ours around the 50th one.

**O(n)** for Linear Search (need to go halfway), **O(log n)** for Binary Search (need to halve the data repeatedly)

**Worst Case :**

The item is found at the very end of the list or is NOT in the list at all. We have to check every item.

Example: We have 100 products, and our target is the 100th one—or doesn’t exist.

**O(n) for Linear search** (need to check every item).

**O(log n)** (need to keep splitting until it’s found or we run out of data).

Now I’ll attach my code and snapshots,

**Main.java:**

package com.example.ecommerce;

public class Main {

    public static void main(String[] args) {

        Product[] products = {

                new Product(1012, "Laptop", "Electronics"),

                new Product(1111, "Leather Jacket", "Fashion"),

                new Product(1500, "Keyboard", "Accessories"),

                new Product(1212, "Running Shoes", "Fashion"),

                new Product(4502, "Monitor", "Electronics")

        };

        // Linear Search

        int targetId = 1212;

        Product foundProductLinear = SearchUtils.linearSearch(products, targetId);

        System.out.println("Linear Search Result for ID " + targetId + " -> " + foundProductLinear);

        SearchUtils.sortProductsById(products);

        // Binary Search

        Product foundProductBinary = SearchUtils.binarySearch(products, targetId);

        System.out.println("Binary Search Result for ID " + targetId + " -> " + foundProductBinary);

    }

}

**Product.java**

package com.example.ecommerce;

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;

    }

    public String getCategory() {

        return category;

    }

    @Override

    public String toString() {

        return "Product{" +

                "productId=" + productId +

                ", productName='" + productName + '\'' +

                ", category='" + category + '\'' +

                '}';

    }

}

**SearchUtils.java**

package com.example.ecommerce;

import java.util.Arrays;

import java.util.Comparator;

public class SearchUtils {

    // Linear search

    public static Product linearSearch(Product[] products, int targetId) {

        for (Product p : products) {

            if (p.getProductId() == targetId) {

                return p;

            }

        }

        return null;

    }

    // Binary search

    public static Product binarySearch(Product[] products, int targetId) {

        int left = 0;

        int right = products.length - 1;

        while (left <= right) {

            int mid = left + (right - left) / 2;

            if (products[mid].getProductId() == targetId) {

                return products[mid];

            } else if (products[mid].getProductId() < targetId) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return null;

    }

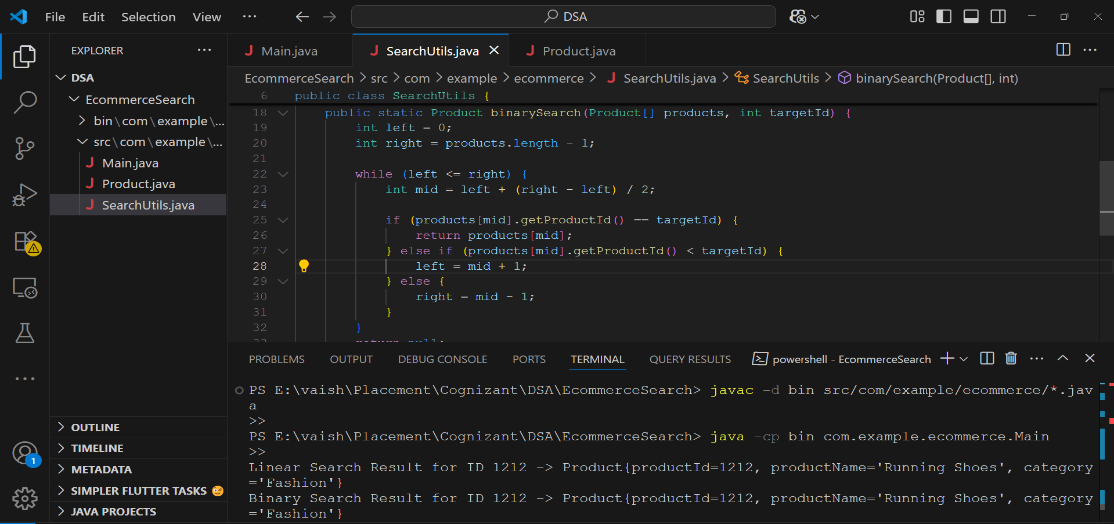
    public static void sortProductsById(Product[] products) {

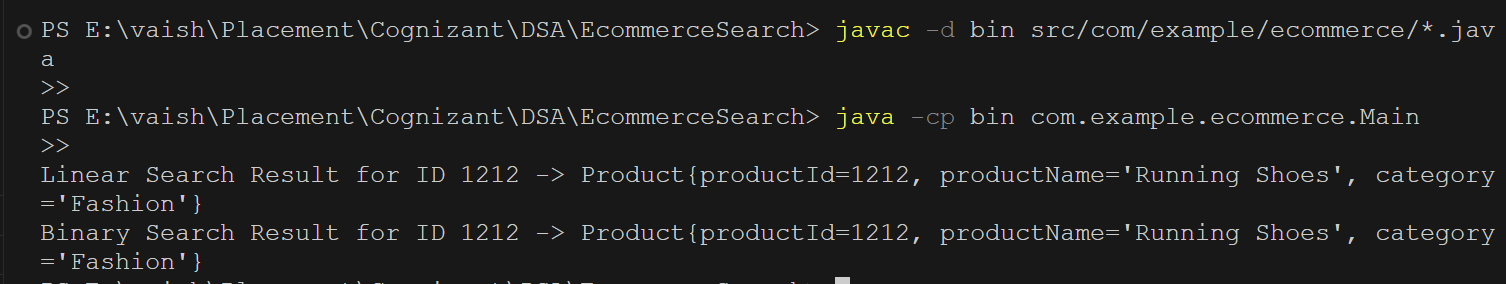
        Arrays.sort(products, Comparator.comparingInt(Product::getProductId));

    }

}

**Output:**





**From my analysis,**

| **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) |

**Which is more suitable for this platform?**

In an e‑commerce platform, the **binary search** is generally **more suitable** when dealing with a large number of products. Here’s why:

**Binary Search**:

* **Works very fast** when the data is sorted (O(log n)), making searches almost instant even with millions of products.
* Ideal for situations like searching a huge online catalog.

**Linear Search**:

* Simple and works well for very small or unsorted data.
* Will be too slow if the number of products is very large (O(n)), because it needs to check every item one by one.