# Algorithms and Data Structures

**Exercise 2: E-commerce Platform Search Function**

1. **Understand Asymptotic Notation:**

Big O notation is a mathematical way to describe how an algorithm’s time or space usage grows as the input size (n) increases. It expresses the upper bound of an algorithm’s complexity — meaning the worst-case scenario.

**It helps developers:**

* Compare the efficiency of different algorithms or data structures.
* Understand the worst-case time or space an algorithm might take.
* Analyze how performance changes as the input size increases.

**Big O Helps in Analyzing Algorithms**

* **Efficiency -** Shows how fast or slow an algorithm runs for different input sizes**.**
* **Scalability -** Helps predict if an algorithm can handle very large data.
* **Optimization -** Highlights areas where performance can be improved.

**Example:**

If searching through 1,000,000 products:

**Linear search (O(n) ) - Linear Time:** Up to 1,000,000 checks . The time increases linearly with input size.

**Binary search (O(log n)) - Logarithmic Time:** Only about 20 checks. The Time increases very slowly

, even with large input. Much faster and more efficient for big datase

So Big O clearly shows binary search is better for large data.

**The best, average, and worst-case scenarios for search operations.**

i) **Linear Search (**O(n)**):**

|  |  |  |  |
| --- | --- | --- | --- |
| | **Scenario** | | --- | | **Description** | **Time Complexity** |
| Best Case | Target is the **first element** in the list | O(1) |
| Average Case | Target is **somewhere in the middle** | |  | | --- | | O(n/2) → O(n) | |
| Worst Case | Target is the **last element** or **not present** | |  | | --- | | O(n) | |

**ii) Binary Search** (O(log n)):

|  |  |  |  |
| --- | --- | --- | --- |
| | **Scenario** | | --- | | Description | Time Complexity |
| Best Case | Target is the **middle element** on the first try | O(1) |
| Average Case | Target found after several splits | |  | | --- | | O(log n) | |
| Worst Case | Target is **not present** after full divide | |  | | --- | | O(log n) | |

1. **Setup:**

*//Product class-* ***Product.java***

public class Product implements Comparable<Product> {

private final int productId;

private final String productName;

private final String category;

public Product(int id, String name, String category) {

this.productId = id;

this.productName = name;

this.category = category;

}

public int getProductId()

{

return productId;

}

public String getProductName()

{

return productName;

}

public String getCategory()

{

return category;

}

@Override

public int compareTo(Product other) {

return Integer.compare(this.productId, other.productId);

}

@Override

public String toString() {

return "%d | %s | %s".formatted(productId, productName, category);

}

}

1. **Implementation:**

*//Search Algorithms -* ***SearchUtil.java***

import java.util.Arrays;

public class SearchUtil {

*//Linear search*

public static int linearSearch(Product[] arr, int key) {

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

if (arr[i].getProductId() == key) {

return i;

}

}

return -1;

}

*//Binary search*

public static int binarySearch(Product[] arr, int key) {

int low = 0, high = arr.length - 1;

while (low <= high) {

int mid = (low + high) >>> 1;

int id = arr[mid].getProductId();

if (id == key)

return mid;

else if (id < key)

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

public static Product[] sortedCopy(Product[] originals) {

Product[] copy = Arrays.copyOf(originals, originals.length);

Arrays.sort(copy);

return copy;

}

}

*//Testing-* ***SearchDemo.java***

public class SearchDemo {

public static void main(String[] args) {

Product[] products = {

new Product(105, "Gaming Mouse", "Electronics"),

new Product(203, "Running Shoes", "Sports"),

new Product(150, "Water Bottle", "Home"),

new Product(302, "Bluetooth Earbuds", "Electronics"),

new Product(112, "Yoga Mat", "Sports")

};

int targetId1 = 150; *// Product exists*

int targetId2 = 999; *// Product does NOT exist*

*// Search for existing product ID*

performSearch(products, targetId1);

*// Search for non-existing product ID*

performSearch(products, targetId2);

}

private static void performSearch(Product[] products, int targetId) {

System.out.println("Searching for Product ID: " + targetId);

*// Linear Search- unsorted list*

int idxLinear = SearchUtil.linearSearch(products, targetId);

System.out.println("Linear Search:");

if (idxLinear != -1)

{

Product found = products[idxLinear];

System.out.printf("Found at index %d:\n Product ID: %d\n Name: %s\n Category: %s\n",

idxLinear, found.getProductId(), found.getProductName(), found.getCategory());

}

else

{

System.out.println("Product not found in the unsorted list.");

}

*// Binary Search - Sorted List*

Product[] sorted = SearchUtil.sortedCopy(products);

int idxBinary = SearchUtil.binarySearch(sorted, targetId);

System.out.println("Binary Search :");

if (idxBinary != -1) {

Product found = sorted[idxBinary];

System.out.printf("Found at sorted index %d:\n Product ID: %d\n Name: %s\n Category: %s\n", idxBinary, found.getProductId(), found.getProductName(), found.getCategory());

}

else

{

System.out.println("Product not found in the sorted list.");

}

System.out.println();

}

}

1. **Analysis:**

**Linear Search – Time Complexity:**

Linear search checks each element in the array one by one until it finds the target or reaches the end.

* As the number of elements n increases, the time taken grows **proportionally**.
* Suitable for **small or unsorted** datasets.
* Not efficient for large datasets.

**Binary Search – Time Complexity:**

Binary search works by repeatedly dividing a **sorted** array in half to locate the target.

* Binary search is **very efficient** for large, sorted datasets.
* Requires the array to be **sorted** beforehand.
* Each step **reduces the search space by half**, making it much faster than linear search.

### Example:

If you have **1,000,000 products**:

* **Linear search** may do up to **1,000,000 comparisons**
* **Binary search** needs at most **20 comparisons** (log₂ 1,000,000 ≈ 20)

**Discuss which algorithm is more suitable for your platform and why.**

For an **e-commerce platform**, ****binary search**** is **more suitable** than linear search.

### ****Performance with Large Data****

* Faster search = better user experience = higher customer satisfaction.
* **Binary Search**: O(log n) → Much faster even with thousands or millions of products.
* **Linear Search**: O(n) → Slower as product catalog grows.

1. **Product Catalog is Often Sorted or Can Be Sorted**

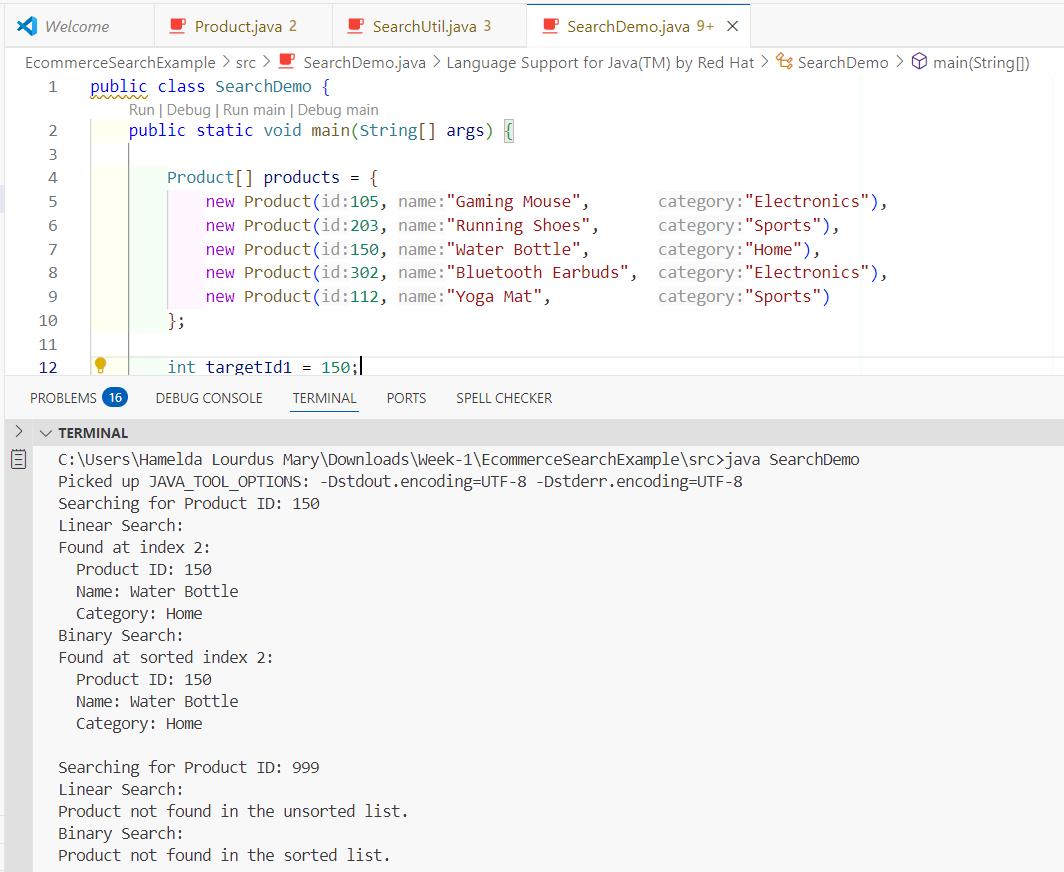
* E-commerce platforms typically store products in databases where data is already **sorted by product ID or name**.
* Binary search **requires sorted data**, which is not a problem in this case.

3. **Frequent Search Operations**

* Search is a **common operation** on shopping platforms.
* Binary search is optimized for frequent, fast lookups.
* Linear search becomes too slow when used repeatedly on large lists.

1. **Scalability:**

* As your product list grows from **hundreds to millions**, binary search scales smoothly.
* Linear search becomes inefficient and affects app performance.



**Output:**

Searching for Product ID: 150

Linear Search :

Found at index 2:

Product ID: 150

Name: Water Bottle

Category: Home

Binary Search :

Found at sorted index 1:

Product ID: 150

Name: Water Bottle

Category: Home

Searching for Product ID: 999

Linear Search :

Product not found in the unsorted list.

Binary Search:

Product not found in the sorted list.

**Exercise 7: Financial Forecasting**

**1 .Understand Recursive Algorithms:**

Recursion is when a method calls **itself** to solve a smaller sub-problem, until it reaches a **base case** that stops the calls.  
It is most helpful when a problem’s definition is **naturally repetitive** or self-similar, e.g.:

* factorial n! = n × (n-1)!
* Fibonacci
* exponentiation (1+r)^n

Similarly , In forecasting, if the value next year depends on this year’s value and a fixed growth rate, you can express it recursively.

**Recursion Simplifies Problems**

* Can be broken down into smaller versions of the same problem.
* Useful for repetitive or branching structure
* Common in problems that follows Mathematical formulas (factorial, Fibonacci)
* Ideal for Tree or graph traversal
* Powers Divide-and-conquer algorithms (e.g., merge sort, quick sort)
* Solves Backtracking problems (e.g., maze solving, sudoku)
* Suitable for Compound interest / financial forecasting with multiple steps
* Can eliminate multiple steps

**In Forecasting Problem**

* Instead of using a loop to apply each year’s growth rate manually, we use recursion
* Automatically keeps track of which year you're in
* Applies growth step-by-step
* Ends neatly when all rates have been processed
* This makes the code simpler and more readable for a problem that naturally has a "repeat until done" structure.

**2.Code:**

//ForecastDemo

import java.util.Arrays;

public class ForecastDemo {

/\*  **STEP 2** –– SETUP

\* Create a method to calculate the future value using a recursive approach.

\*/

public static double calculateFutureValueRecursive(double presentValue, double[] annualGrowthRates, int currentYearIndex) {

// Base Case - no more years left

if (annualGrowthRates == null || annualGrowthRates.length == 0) {

return presentValue;

}

        if (currentYearIndex == annualGrowthRates.length ) {

            return presentValue;

        }

/\* **STEP 3 –– IMPLEMENTATION**

\* Implement a recursive algorithm to predict future values based on past growth rates.

\* Recursive algorithm: Apply this year’s growth, then call the method for next year.

\*/

 double updatedValue = presentValue \* (1 + annualGrowthRates[currentYearIndex]);

      return calculateFutureValueRecursive(updatedValue, annualGrowthRates, currentYearIndex + 1);

}

/\* **OPTIMIZATION** – To improve performance and avoid excessive computation.

\* - Avoids deep recursion and stack overhead.

\* - An iterative version performs the same O(n) work but uses only O(1) extra space,

\* eliminating the risk of StackOverflowError for very long rate histories.

\*/

  public static double calculateFutureValueIterative(double presentValue, double[] annualGrowthRates) {

        double futureValue = presentValue;

        for (double yearlyRate : annualGrowthRates) {

            futureValue = futureValue \* (1 + yearlyRate);

        }

        return futureValue;

}

public static void main(String[] args) {

        double initialInvestment = 10000.0;

        double[] growthRatesPerYear = {0.05, 0.06, 0.04, 0.07, 0.03}; // 5%, 6%, 4%, 7%, 3%

      double resultRecursive = calculateFutureValueRecursive(initialInvestment, growthRatesPerYear, 0);

      double resultIterative = calculateFutureValueIterative(initialInvestment, growthRatesPerYear);

        System.out.println("=== Financial Forecasting Using Past Growth Rates ===\n");

        System.out.printf("Initial Investment       : $%.2f%n", initialInvestment);

        System.out.println ("Annual Growth Rates      : " + Arrays.toString(growthRatesPerYear));

        System.out.printf("Total Years Forecasted   : %d%n%n", growthRatesPerYear.length);

        System.out.println("--- Forecast Results ---");

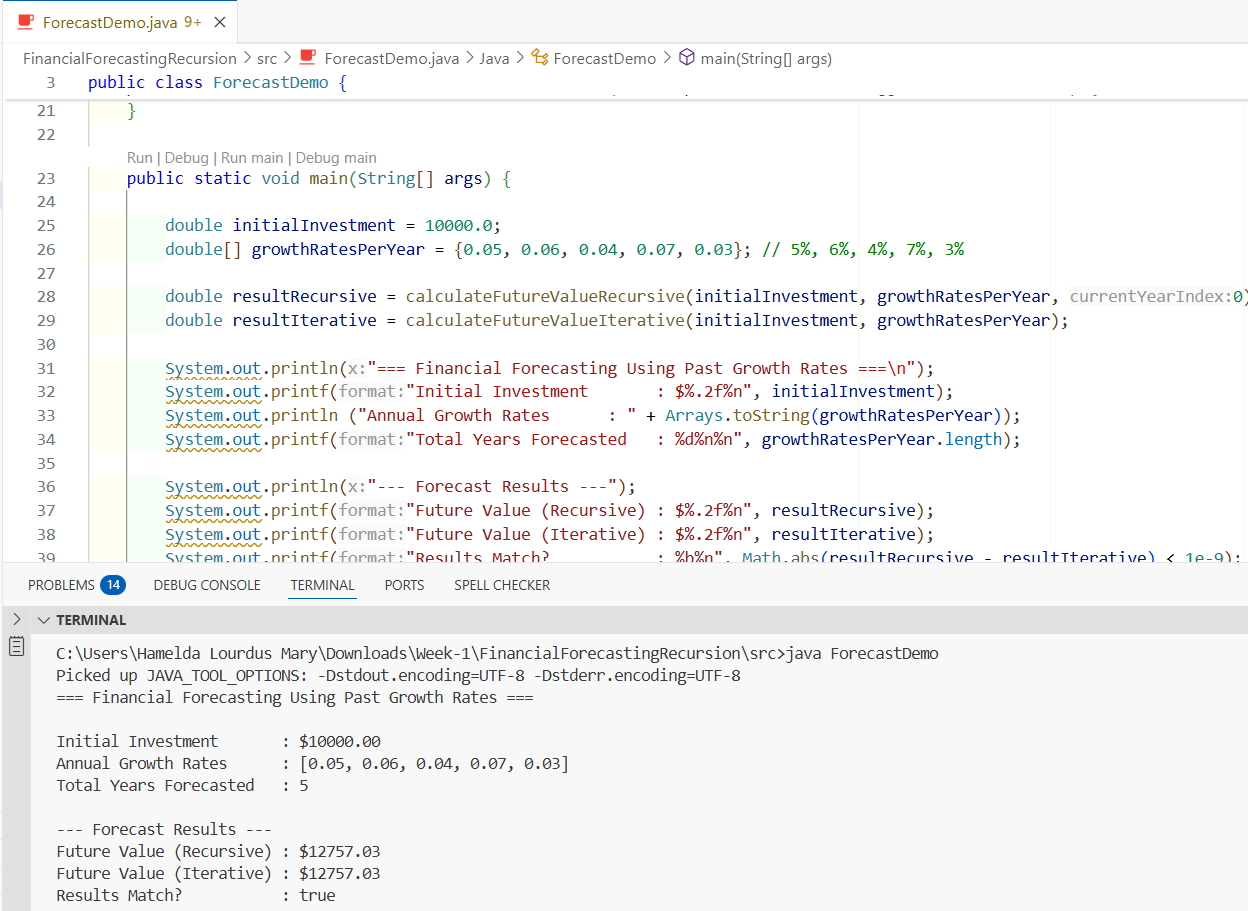
        System.out.printf("Future Value (Recursive) : $%.2f%n", resultRecursive);

        System.out.printf("Future Value (Iterative) : $%.2f%n", resultIterative);

    System.out.printf("Results Match?           : %b%n", Math.abs(resultRecursive - resultIterative) < 1e-9);

}

}



***Output:***

=== Financial Forecasting Using Past Growth Rates ===

Initial Investment : $10000.00

Annual Growth Rates : [0.05, 0.06, 0.04, 0.07, 0.03]

Total Years Forecasted : 5

--- Forecast Results ---

Future Value (Recursive) : $12757.03

Future Value (Iterative) : $12757.03

Results Match? : true

1. **Analysis:**

**Time Complexity = O(n),** where n = number of past growth rates.

* If there are n years (i.e. n = rates.length), then the function makes n **recursive calls**. The method is called **once per year**.
* Each call takes **constant time** O(1).

**Space Complexity = O(n)**

Because it’s **recursive**, it also uses **stack memory**:

* Each recursive call adds one frame to the call stack
* So, **stack depth = n**
* You can reduce this to **O(1)** space by converting the algorithm to an **iterative loop**.
* Explain how to optimize the recursive solution to avoid excessive computation.

**To optimize the recursive solution to avoid excessive computation:**

**Convert to Iterative:**

To **avoid recursion and reduce memory usage**, convert it into a loop:

//optimized

 public static double calculateFutureValueIterative(double presentValue, double[] annualGrowthRates) {

        double futureValue = presentValue;

        for (double yearlyRate : annualGrowthRates) {

            futureValue = futureValue \* (1 + yearlyRate);

        }

        return futureValue;

}

**Time Complexity :** O(n)

**Space Complexity :** O(1)

***General Optimization Techniques:***

**Memoization:** O(n)- Much faster, no repeated work

* Cache results of previous calls to avoid redundant computation (useful for overlapping subproblems like Fibonacci).
* Convert to Iteration:O(n)- Fastest, constant memory
* Recursion can often be rewritten as loops to reduce call stack overhead.

**Limit Depth:**

Add early-exit or maximum-depth checks for safety in deeply recursive cases.

**Use Logarithmic Algorithms**

Where applicable, use binary recursion (like exponentiation by squaring) to reduce recursive depth to O(log n).