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

Big O notation is a powerful tool used in computer science to describe the time complexity or space complexity of algorithms. Big-O is a way to express the upper bound of an algorithm’s time or space complexity**.**

Big O is part of asymptotic analysis, which is used to estimate how much time or memory an algorithm needs to solve a problem especially as the input size becomes very large.

* **Predicts performance at scale**: It lets you understand how an algorithm behaves when handling large inputs, making sure the software stays fast as data grows.
* **Saves time and effort**: Instead of testing every time the system changes, you can use Big O to evaluate efficiency in a more consistent, theoretical way.
* **Compares multiple solutions**: It allows you to choose the best algorithm by comparing their growth patterns (like linear vs. quadratic time).
* **Builds scalable software**: Algorithms with better Big O performance help ensure your application handles real-world loads smoothly.

**Best Case**

This is the scenario where the search operation completes with the least effort.

* For **linear search**, the best case happens when the target element is the first item in the list.
* Time complexity: **O(1)** constant time.

**Average Case**

This represents the expected performance over all possible inputs.

* For linear search, if we assume the element is equally likely to be anywhere (or not present), the average case is when it’s found somewhere in the middle.
* Time complexity: **O(n/2)**.

**Worst Case**

This is the scenario where the search takes the *most time*.

* For linear search, this happens when the element is *not in the list* or is the *last item*.
* Time complexity: **O(n)** linear time.

**Product.java:**

package EcommercePlatormSearching;

public class Product {

int productId;

String productName;

String category;

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

this.productId = id;

this.productName = name;

this.category = category;

}

}

**Searching.java:**

package EcommercePlatormSearching;

import java.util.Arrays;

import java.util.Comparator;

public class Searching {

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

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

if (products[i].productName.equalsIgnoreCase(name)) {

return i;

}

}

return -1;

}

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

Arrays.*sort*(products, Comparator.*comparing*(p -> p.productName));

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

int cmp = products[mid].productName.compareToIgnoreCase(name);

if (cmp == 0)

return mid;

else if (cmp < 0)

left = mid + 1;

else

right = mid - 1;

}

return -1;

}

}

**SearchTest.java:**

package EcommercePlatormSearching;

public class SearchTest {

public static void main(String[] args) {

Product[] products = {

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

new Product(102, "Shirt", "Clothing"),

new Product(104, "Phone", "Electronics"),

new Product(103, "Book", "Education")

};

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

int result1 = Searching.*linearSearch*(products, "book");

System.*out*.println(result1 != -1 ? "Found at index : " + result1 : "Not found");

System.*out*.println("\nBinary Search:");

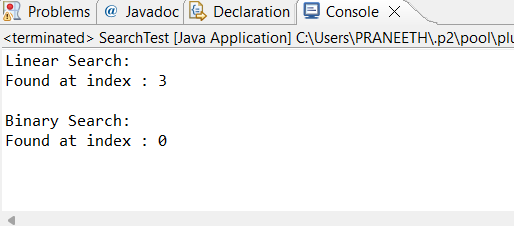
int result2 = Searching.*binarySearch*(products, "BOOK");

System.*out*.println(result2 != -1 ? "Found at index : " + result2 : "Not found");

}

}

**Output:**



**Time Complexity Analysis**

* **Linear Search**: O(n) - Scans every element; better for small or unsorted data.
* **Binary Search**: O(log n) - Requires sorted data; significantly faster for large datasets.
* For an e-commerce platform with thousands of products, **Binary Search** is preferred after sorting the data.

**Exercise 7: Financial Forecasting**

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. It continues doing this until it reaches a base case, which is a condition that stops the recursion.

**How It Simplifies Problems:**

* Breaks down complex tasks into smaller, manageable subproblems.
* Reduces code duplication, especially in problems with repetitive patterns (like calculating growth over time).
* Matches natural problem structure, such as in mathematical formulas (e.g., factorial, Fibonacci, compound interest).
* Improves readability, making the logic easier to follow when the recursive pattern is clear.

**FutureValue.java:**

package FinancialForecasting;

public class FutureValue {

public static double predictFutureValue(double initialValue, double growthRate, int years) {

if (years == 0) {

return initialValue;

}

return *predictFutureValue*(initialValue \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

double initial = 10000.0;

double rate = 0.10;

int years = 3;

System.*out*.println("Recursive Prediction:");

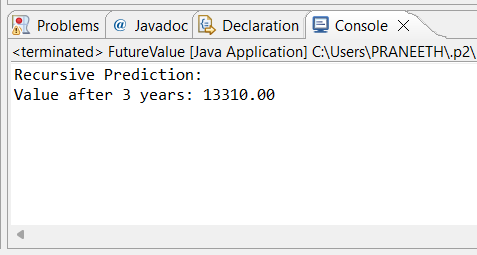
double recursiveResult = *predictFutureValue*(initial, rate, years);

System.*out*.printf("Value after %d years: %.2f%n", years, recursiveResult);

}

}

**Output:**

****

**Time Complexity & Optimization:**

The recursive method runs once per year, so its time complexity is O(n), where n is the number of years.

For simple linear recursion like this, optimization isn’t critical.

However, for more complex recursive models (e.g., involving multiple branches like Fibonacci), use memoization or convert to iteration to avoid redundant calculations and stack overflow.

**Memoized Recursive Forecasting:**

import java.util.HashMap;

import java.util.Map;

public class FutureValue {

private static Map<Integer, Double> memo = new HashMap<>();

public static double predictFutureValue(double initialValue, double growthRate, int years) {

if (years == 0) {

return initialValue;

}

if (memo.containsKey(years)) {

return memo.get(years) \* (initialValue / baseValue);

}

double future = predictFutureValue(initialValue \* (1 + growthRate),growthRate,years-1);

memo.put(years, future / Math.pow(1 + growthRate, years - 1));

return future;

}

}