**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.

How Big O helps in analyzing algorithms:

**1. Comparing Algorithms:**

Big O notation allows for a standardized comparison of different algorithms' efficiency, helping in selecting the best one for a specific task or dataset.

**2. Predicting Scalability:**

By understanding how an algorithm's runtime or memory usage scales, developers can predict how it will perform with larger inputs and make informed decisions about its suitability for different scenarios.

**3. Optimizing Performance:**

Big O helps identify potential bottlenecks in algorithms, enabling developers to optimize performance by focusing on the most critical parts of the code.

**4. Choosing the Right Data Structures:**

Big O notation helps in selecting appropriate data structures that offer optimal performance for specific operations.

Best, average, and worst-case scenarios for search operations:

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Linear Search** | **Binary Search** |
| **Best** | O(1) | O(1) (middle match) |
| **Average** | O(n/2) | O(log n) |
| **Worst** | O(n) | O(log n) |

**Product.java:**

package Searching;

public 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;

}

@Override

public String toString() {

return productId+": "+productName+","+category;

}

}

**LinearSearch.java:**

package Searching;

public class LinearSearch {

public static Product search(Product[] products,String productName) {

for(Product product:products){

if(product.productName.equals(productName)){

return product;

}

}

return null;

}

}

**BinarySearch.java:**

package Searching;

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

public static void sortProducts(Product[] products){

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

}

public static Product search(Product[] products,String productName){

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

while(left<=right){

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

int compare=products[mid].productName.compareTo(productName);

if(compare==0)

return products[mid];

else if(compare<0)

left=mid+1;

else

right=mid-1;

}

return null;

}

}

**Main.java:**

package Searching;

public class Main {

public static void main(String[] args) {

Product[] products={

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

new Product(102,"Shoes","Fashion"),

new Product(103,"Smartphone","Electronics"),

new Product(104,"Notebook","Stationery")

};

Product result1=LinearSearch.*search*(products,"Shoes");

System.*out*.println("Linear Search: "+result1);

Product result2=LinearSearch.*search*(products,"Tab");

System.*out*.println("Linear Search: "+result2);

BinarySearch.*sortProducts*(products);

Product result3=BinarySearch.*search*(products,"Notebook");

System.*out*.println("Binary Search: "+result3);

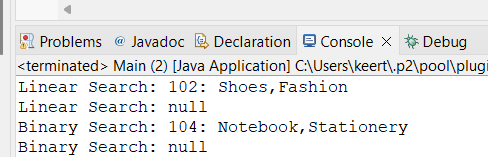
Product result4=BinarySearch.*search*(products,"Watch");

System.*out*.println("Binary Search: "+result4);

}

}

**Output:**



Linear Search is simple and requires no sorting, but is slow for large datasets.

Binary Search is much faster (especially for large product catalogs), but requires:

* Sorted data
* Extra preprocessing time to sort

For E-Commerce use Binary Search for performance if the product list is sorted or indexed.

**Exercise 7: Financial Forecasting**

**Recursion** is when a function calls itself to solve a problem by breaking it into smaller sub-problems.

It is used because it simplifies problems with repetitive or nested patterns, like:

* Calculating future values over time
* Navigating trees
* Computing Fibonacci-like sequences

**Forecasting Future Value:**

Assume:

* There is a current value.
* Company has a fixed annual growth rate.
* You want to know the value after n years.

**FinancialForecast.java:**

package FinancialForecast;

public class FinancialForecast {

public static double forecast(double currentValue,double growthRate,int years) {

if(years==0){

return currentValue;

}

return *forecast*(currentValue\*(1+growthRate),growthRate,years-1);

}

public static void main(String[] args) {

double currentValue=1000.0;

double annualGrowthRate=0.05;

int years = 5;

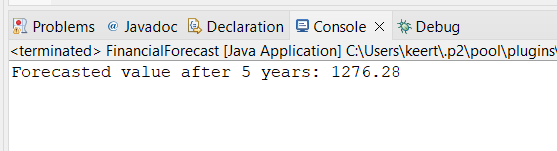
double future=*forecast*(currentValue,annualGrowthRate,years);

System.*out*.printf("Forecasted value after %d years: %.2f%n",years,future);

}

}

**Output:**



**Time Complexity:**

* Recursive depth = years
* So Time Complexity = O(n)

**Space Complexity:**

* Due to recursion stack: O(n)

**Optimization:**

To avoid stack overflow in deep recursion (e.g., years = 10000), use iteration or tail recursion.

**Iterative Version:**

public static double forecastIterative(double currentValue, double growthRate, int years) {

for (int i = 0; i < years; i++) {

currentValue \*= (1 + growthRate);

}

return currentValue;

}

This version uses O(1) space and is more efficient for large inputs.