**Exercise 2: E-commerce Platform Search Function (Mandatory)**

**Understand Asymptotic Notation:**

Q) Explain Big O notation and how it helps in analyzing algorithms.

* It is a mathematical way to describe the time and space complexity of an algorithm.
* It tells how the algorithm behaves as the input size changes.

Q) Describe the best, average, and worst-case scenarios for search operations.

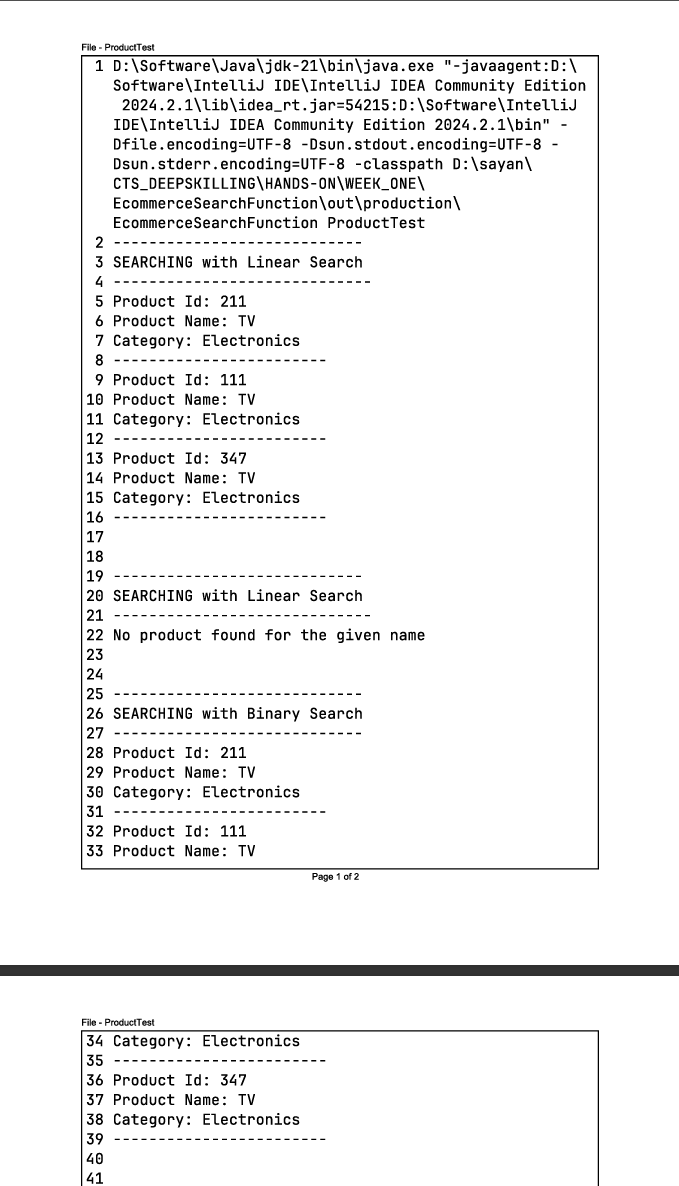
* **Linear Search:**
  + **Best Case**: This occurs when the searched element **is present at the beginning** of the array. Time Complexity: O(1).
  + **Average Case**: This occurs when the target is **somewhere in the middle.** Time Complexity: O(N).
  + **Worst Case:** This occurs when the target is the **last** element or **not present at all.** Time Complexity: O(N).
* **Binary Search:**
  + **Best Case**: This occurs when the target is present at the middle of the array on **first try**. Time Complexity: O(1).
  + **Average Case**: This occurs when the target is found at the middle of the array after **several divisions**. Time Complexity: O(log n).
  + **Worst Case:** This occurs when the target is **not present or found at the deepest level.** Time Complexity: O(log n).

**PROGRAM:**

*import* java.util.ArrayList;  
*import* java.util.List;

//test class  
*public class* ProductTest {  
 *public static void* main(String[] args) {  
 *//creating only one object of class* Product productObj = Product.getInstance();  
  
 *//adding products* productObj.addProduct(211,"TV", "Electronics");  
 productObj.addProduct(131,"Biscuits", "Food");  
 productObj.addProduct(76,"Table", "Furniture");  
 productObj.addProduct(456,"Earphones", "Electronics");  
  
 *//adding similar products but different ids* productObj.addProduct(111,"TV", "Electronics");  
 productObj.addProduct(347,"TV", "Electronics");  
 productObj.addProduct(178,"Table", "Furniture");  
  
 *//searching for a product using linear search* productObj.LinearSearchWithName("tv");  
  
 *//For better output view* System.out.println();  
 System.out.println();  
  
 productObj.LinearSearchWithName("egg"); *//will show product not found  
  
 //sorting to apply binary search* productObj.sortProductsByName();  
  
 *//For better output view* System.out.println();  
 System.out.println();  
  
 *//finding all possible product having the same name using binary search* productObj.BinarySearchWithName("tv");  
  
 *//For better output view* System.out.println();  
 System.out.println();  
  
 productObj.BinarySearchWithName("cutlery");  
  
  
 }  
}

//class of products  
*class* Product{  
  
 *private static* Product instance = *null*;  
  
 *private* List<Integer> productId;  
 *private* List<String> productName;  
 *private* List<String> category;  
  
 *private* Product(){  
 productId = *new* ArrayList<>();  
 productName = *new* ArrayList<>();  
 category = *new* ArrayList<>();  
 }  
  
 *public static* Product getInstance(){  
 *if*(instance == *null*){  
 instance = *new* Product();  
 }  
 *return* instance;  
 }  
  
 *public void* addProduct(*int* productId, String productName, String category){  
 *//adding the id  
 this*.productId.add(productId);  
 *//adding corresponding product in the same index as id  
 this*.productName.add(productName);  
 *//adding the category  
 this*.category.add(category);  
 }  
  
 *public void* LinearSearchWithName(String productName){  
  
 *boolean* productFound = *false*;  
 System.out.println("----------------------------");  
 System.out.println("SEARCHING with Linear Search");  
 System.out.println("-----------------------------");  
 *for*(*int* i = 0; i<*this*.productName.size();i++){  
 *if*(*this*.productName.get(i).equalsIgnoreCase(productName)){  
 productFound = *true*;  
 System.out.println("Product Id: " + *this*.productId.get(i));  
 System.out.println("Product Name: " + *this*.productName.get(i));  
 System.out.println("Category: "+ *this*.category.get(i));  
 System.out.println("------------------------");  
 }  
 }  
  
 *if*(!productFound)  
 System.out.println("No product found for the given name");  
 }  
  
 *//to sort the products by name before applying binary search  
 public void* sortProductsByName(){  
  
 *for* (*int* i = 0; i < *this*.productName.size()-1; i++) {  
 *for*(*int* j = 0; j<*this*.productName.size()-1-i;j++){  
 *if*(*this*.productName.get(j).compareToIgnoreCase(*this*.productName.get(j+1)) > 0 ){  
  
 *//swapping the corresponding ids keeping the index same  
 int* tempId = *this*.productId.get(j);  
 *this*.productId.set(j,*this*.productId.get(j+1));  
 *this*.productId.set(j+1,tempId);  
  
 *//swapping the products* String tempProduct = *this*.productName.get(j);  
 *this*.productName.set(j,*this*.productName.get(j+1));  
 *this*.productName.set(j+1,tempProduct);  
  
 *//swapping the categories* String tempCategory = *this*.category.get(j);  
 *this*.category.set(j,*this*.category.get(j+1));  
 *this*.category.set(j+1,tempCategory);  
  
  
 }  
 }  
 }  
*// for (int i = 0; i < this.productId.size(); i++) {  
// System.out.println(this.productId.get(i));  
// }* }  
  
  
 *public void* BinarySearchWithName(String productName){  
 *int* start = 0;  
 *int* end = *this*.productName.size()-1;  
 *int* mid = start + (end - start)/2;  
 *boolean* productFound = *false*;  
  
 System.out.println("----------------------------");  
 System.out.println("SEARCHING with Binary Search");  
 System.out.println("----------------------------");  
  
 *while*(start < end) {  
 *if* (*this*.productName.get(mid).compareToIgnoreCase(productName) > 0) {  
  
 end = mid - 1;  
  
 } *else if* (*this*.productName.get(mid).compareToIgnoreCase(productName) < 0){  
  
 start = mid + 1;  
 }  
 *else*{  
 productFound = *true*;  
  
 *//to find the first index of the element* end = mid-1;  
  
  
  
 }  
  
 mid = start + (end - start)/2;  
 }  
  
 *//to print out all the similar elements  
 while*(*this*.productName.get(mid).compareToIgnoreCase(productName) == 0){  
 System.out.println("Product Id: " + *this*.productId.get(mid));  
 System.out.println("Product Name: " + *this*.productName.get(mid));  
 System.out.println("Category: "+ *this*.category.get(mid));  
 System.out.println("------------------------");  
  
 mid++;  
 *//to prevent overflow  
 if*(mid == *this*.productName.size()) *break*;  
 }  
  
  
 *if*(!productFound)  
 System.out.println("No product found for the given name");  
 }  
  
}

**OUTPUT:**

**ANALYSIS:**

1. **Compare the time complexity of linear and binary search algorithms.**

* Time complexity of Linear Search in program:
* Best case occurs when the **first element** is the required element in the array **productName.**
* Worst case occurs when the required element is found (or not found) after searching the entire array.

Time Complexity: **O(N)**

* Time complexity of Binary Search in program:
* Best case occurs when the required element is exactly at the middle of the array **productName.**
* In average case the time complexity is **O(log n).**

**Extra loop:**

To display all the matching conditions of the item searched, the loop takes **O(K)** time complexity where K is the **number of matching items**. So,

Time Complexity: **O(log n + K)**

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

Binary Search will be better in this case because it works on the principle of divide and conquer. So, on average it takes O(log n + K) time complexity.

If the value of K is large it reaches the worst case scenario O(K) which is equivalent to the linear search algorithm.

**Hence, Binary search is better than Linear Search in average case.**

**Exercise 7: Financial Forecasting (Mandatory)**

**UNDERSTANDING RECURSIVE ALGORITHM:**

1. **Explain the concept of recursion and how it can simplify certain problems.**

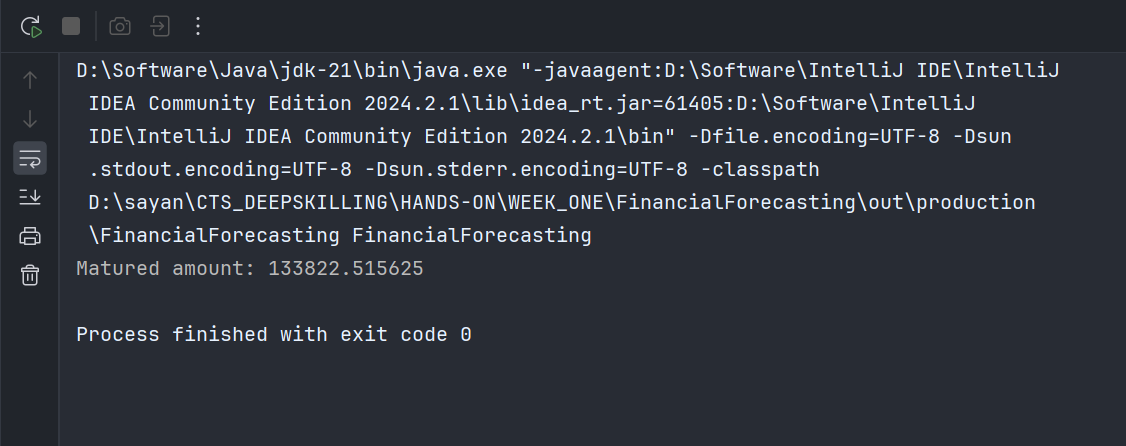
Recursion is the process by which a function calls itself to perform certain computations. Internally, recursion uses stack data structure.

Recursion helps in certain cases like in merge sort where recursion helps in diving the array into multiple subarrays. It also helps in problems which include backtracking as well.

**PROGRAM:**

*public class* FinancialForecasting {  
  
 *//creating a method to predict futureValue  
 private static double* futureValue(*float* principal, *int* interest,*int* years){  
 *//check if maturity date has arrived(Base case)  
 if*(years == 0)  
 *return* principal;  
  
 *//calculate amount with interest for the given year  
 float* futureAmount = principal \*(1+ (*float*)interest/100);  
  
 *//recursive call to generate future value  
 return* futureValue(futureAmount,interest,years-1);  
  
 }  
  
 *public static void* main(String[] args) {  
 *float* principal = 100000.0f;  
 *int* interest = 6;  
 *int* years = 5;  
  
 System.out.println("Matured amount: "+ futureValue(principal,interest,years));  
 }  
}

**OUTPUT:**

****

**ANALYSIS:**

1. **Discuss the time complexity of your recursive algorithm.**

The recursive call occurs **N** times where **N** is the number of years taken to get matured value.

So the overall time complexity is **O(N)**.

**Q) Explain how to optimize the recursive solution to avoid excessive computation.**

In this program, the recursive solution can be optimized by directly using the formula of compounding in a function as give below –

*private static double futureValue(float principal, int interest,int years){  
   
 return principal \* Math.pow(1+(float)interest/100,years);  
  
}*

The above code implements the computation in O(1) time and requires no usage of stack, thereby having O(1) space complexity as well. It, thus, has **no issues of stack overflow error** when large value of years is provided.