**1.E-commerce Platform Search Function**

**What is Big O Notation?**

Big O notation describes the upper bound of an algorithm’s running time (or space requirement) as the input size grows. It gives an abstract measure of how an algorithm behaves in the worst case, regardless of machine or programming language. It focus on the most basic operations of an algorithm.

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

**Best Case:** The target item is found immediately (e.g., first position).

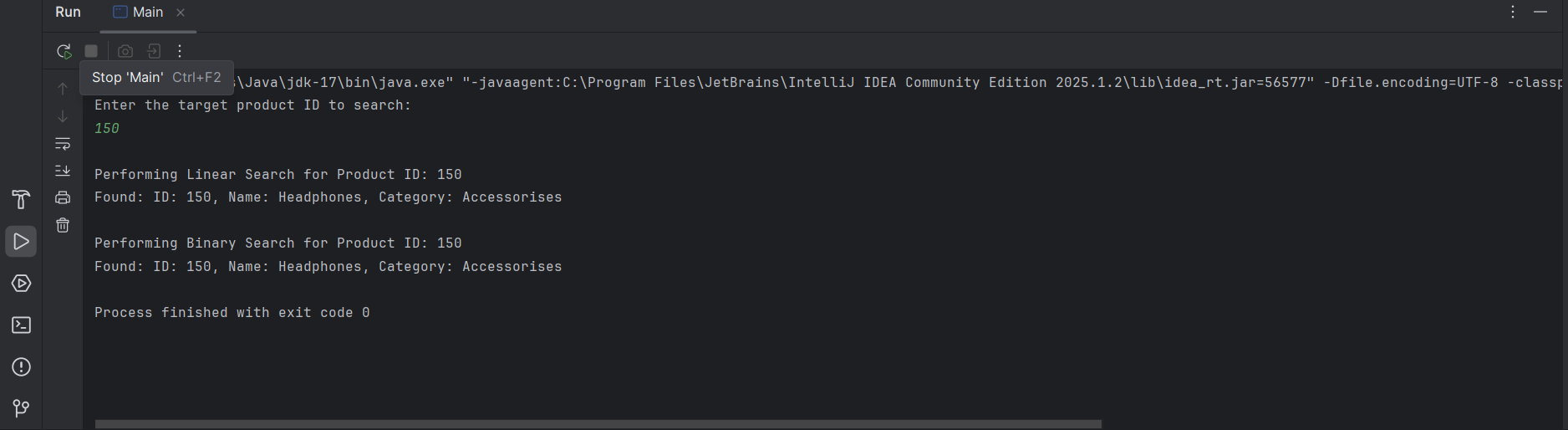
**Average Case:** The target is found roughly halfway through the data.

**Worst Case:** The target is found at the end (or not found), requiring a complete traversal of the data.

**CODE:**

import java.util.\*;  
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;  
 }  
 @Override  
 public String toString()  
 {  
 return "ID: " + productId + ", Name: " + productName + ", Category: " + category;  
 }  
  
}  
public class Main  
{  
 public static Product linearSearch(Product[] products, int targetId)  
 {  
 for (Product p : products) {  
 if (p.productId == targetId) {  
 return p;  
 }  
 }  
 return null;  
 }  
 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].productId == targetId) {  
 return products[mid];  
 } else if (products[mid].productId < targetId) {  
 left = mid + 1;  
 } else {  
 right = mid - 1;  
 }  
 }  
  
 return null;  
 }  
 public static void main(String[] args)  
 {  
 Scanner sc=new Scanner(System.*in*);  
 System.*out*.println("Enter the target product ID to search:");  
 int targetId=sc.nextInt();  
 Product[] products = {  
 new Product(101, "Laptop", "Electronics"),  
 new Product(305, "Phone", "Electronics"),  
 new Product(210, "Shoes", "Footwear"),  
 new Product(150, "Headphones", "Accessories"),  
 new Product(250, "Watch", "Accessories")  
 };  
 System.*out*.println("\nPerforming Linear Search for Product ID: " + targetId);  
 Product result1 = *linearSearch*(products, targetId);  
 if (result1 != null) {  
 System.*out*.println("Found: " + result1);  
 } else {  
 System.*out*.println("Product Not Found");  
 }  
 Arrays.*sort*(products, Comparator.*comparingInt*(p -> p.productId));  
 System.*out*.println("\nPerforming Binary Search for Product ID: " + targetId);  
 Product result2 = *binarySearch*(products, targetId);  
 if (result2 != null) {  
 System.*out*.println("Found: " + result2);  
 } else {  
 System.*out*.println("Product Not Found");  
 }  
 }  
}

**OUTPUT:**



**ANALYSIS:**

**Linear search:**

Linear search time complexity is O(n). It is good for small and unsorted arrays.

**Binary search:**

Binary search time complexity is O(log n). It is very efficient for larger and sorted arrays.

For E-commerce Platform, with a large number of products, Binary Search is preferred for efficiency.

**2.Financial Forecasting**

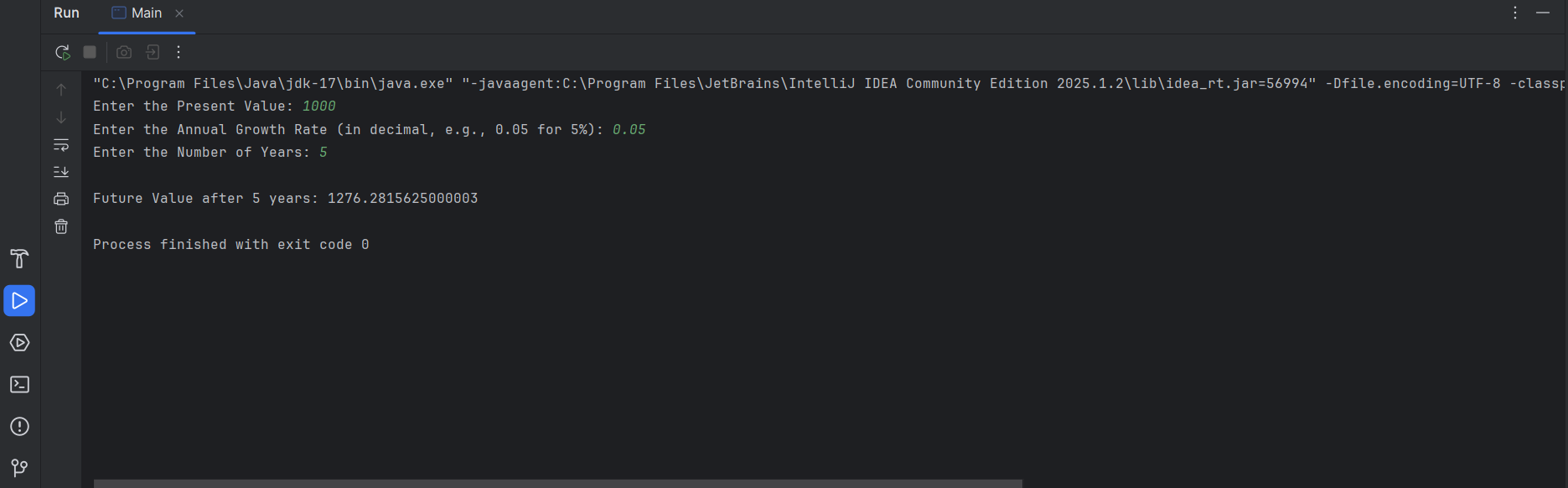
**Recursion:**

Recursion is when a method calls itself to solve a smaller version of the same problem.  
It’s used when a problem can be broken down into smaller, similar sub-problems.

**CODE:**

import java.util.\*;  
  
 class FinancialForecast {  
  
  
 public static double futureValue(double presentValue, double growthRate, int years) {  
  
 if (years == 0) {  
 return presentValue;  
 }  
  
 return *futureValue*(presentValue, growthRate, years - 1) \* (1 + growthRate);  
 }  
 }  
public class Main {  
 public static void main(String[] args) {  
 Scanner sc = new Scanner(System.*in*);  
   
 System.*out*.print("Enter the Present Value: ");  
 double presentValue = sc.nextDouble();  
  
 System.*out*.print("Enter the Annual Growth Rate (in decimal, e.g., 0.05 for 5%): ");  
 double growthRate = sc.nextDouble();  
  
 System.*out*.print("Enter the Number of Years: ");  
 int years = sc.nextInt();  
   
 double result = FinancialForecast.*futureValue*(presentValue, growthRate, years);  
 System.*out*.println("\nFuture Value after " + years + " years: " + result);  
 }  
}

**OUTPUT:**

****

**ANALYSIS:**

**Time Complexity:**

Time Complexity = O(n) where n is the number of years.

If n is very large it may result it stack overflow.

**Optimization:**

Use an iterative approach (loop) to compute the result.