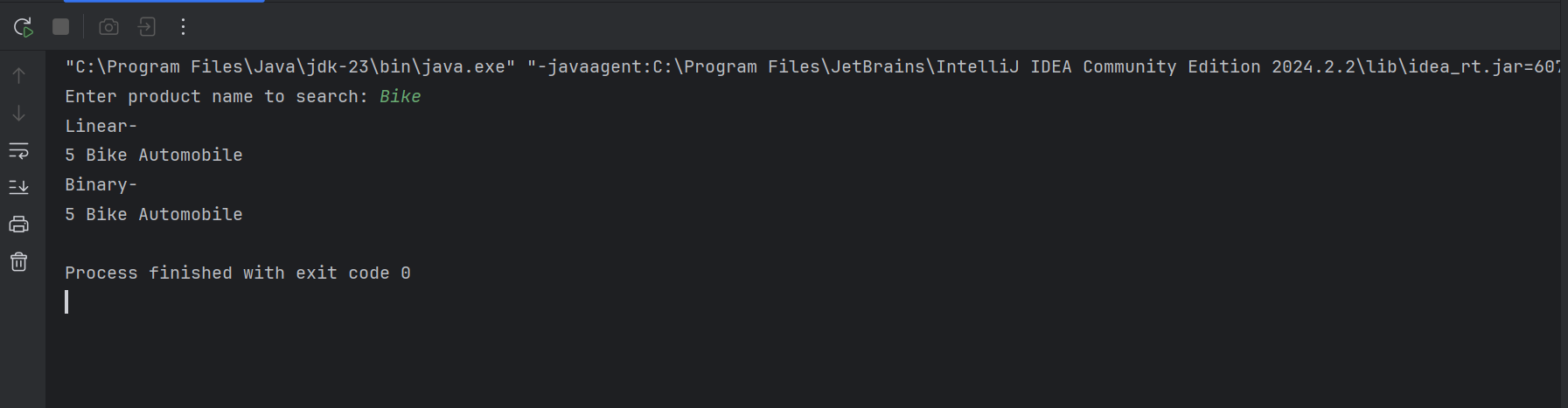
Java FSE Week-1

(Data Structure)

**Q1 E-commerce Platform-**

**//Product class**  
  
public class Product {  
 private int productId;  
 private String productName;  
 private String category;  
  
 public Product(int productId, String productName, String category) {  
 this.productId = productId;  
 this.productName = productName;  
 this.category = category;  
 }  
  
 public int getProductbyId() {  
 return productId;  
 }  
  
 public String getProductbyName() {  
 return productName;  
 }  
  
 public String getbyCategory() {  
 return category;  
 }  
  
 public void showDetails() {  
 System.*out*.println(productId + " " + productName + " " + category);  
 }  
}  
  
  
**// ProductSearch class**

import java.util.\*;  
  
public class ProductSearch {  
  
 public static Product linearSearch(List<Product> products, String name) {  
 for (Product p : products) {  
 if (p.getProductbyName().equalsIgnoreCase(name)) {  
 return p;  
 }  
 }  
 return null;  
 }  
  
 public static Product binarySearch(List<Product> products, String name) {  
 int low = 0, high = products.size() - 1;  
  
 while (low <= high) {  
 int mid = (low + high) / 2;  
 Product midProduct = products.get(mid);  
 int cmp = midProduct.getProductbyName().compareToIgnoreCase(name);  
  
 if (cmp == 0) {  
 return midProduct;  
 } else if (cmp < 0) {  
 low = mid + 1;  
 } else {  
 high = mid - 1;  
 }  
 }  
  
 return null;  
 }  
  
 public static void main(String[] args) {  
 List<Product> productList = new ArrayList<>();  
 productList.add(new Product(1, "Nintendo", "Gaming Electronics"));  
 productList.add(new Product(2, "DEll- Inspiron", "Electronics"));  
 productList.add(new Product(3, "Shirts", "Clothing"));  
 productList.add(new Product(4, "PS-5", "Gaming Electronics"));  
 productList.add(new Product(5, "Bike", "Automobile"));  
 productList.add(new Product(5, "Game-Box", "Gaming Electronics"));  
  
 productList.sort(Comparator.*comparing*(Product::getProductbyName));  
  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.print("Enter product name to search: ");  
 String inputName = scanner.nextLine();  
  
 Product foundLinear = *linearSearch*(productList, inputName);  
 if (foundLinear != null) {  
 System.*out*.println("Linear-");  
 foundLinear.showDetails();  
 } else {  
 System.*out*.println("Product not found");  
 }  
  
 Product foundBinary = *binarySearch*(productList, inputName);  
 if (foundBinary != null) {  
 System.*out*.println("Binary-");  
 foundBinary.showDetails();  
 } else {  
 System.*out*.println("Product not found");  
 }  
 }  
}



Big O Notation

Big O notation is a way for us to talk about how efficient an algorithm is—specifically, how its performance (in terms of time or space) changes as the size of the input grows. It gives us a general idea of the worst-case scenario for how long an algorithm might take or how much memory it might need. This is really useful when we want to compare different approaches and pick the one that scales best with large data.

Let’s walk through the most common Big O notations and what they mean:

O(1) – Constant Time:  
This means the algorithm always takes the same amount of time, no matter how big the input is. A good example is directly accessing an item in an array using its index.

O(log n) – Logarithmic Time:  
Here, the time grows slowly even as the input gets large. Binary search is a classic example—it keeps cutting the data in half each step, making it super efficient for sorted arrays.

O(n) – Linear Time:  
The time grows directly with the size of the input. If you have to look through each item once, like in a simple linear search, that’s O(n).

O(n log n) – Linearithmic Time:  
This is a mix of linear and logarithmic growth. Sorting algorithms like mergesort and heapsort fall into this category. They’re quite efficient for large datasets.

O(n²) – Quadratic Time:  
Time grows quickly with input size—especially with nested loops. Bubble sort and insertion sort are common examples, and they can be slow on large inputs.

O(2ⁿ) – Exponential Time:  
These algorithms double in time with every new element added. Problems like the brute-force solution to the traveling salesman problem fall into this category and become impractical very quickly.

O(n!) – Factorial Time:  
The slowest of the bunch—time increases factorially with input size. This is what you get when checking every possible arrangement or permutation, like in some brute-force combinatorial problems.

Big O notation is essential when evaluating and comparing algorithms. It helps us:

Understand how an algorithm behaves as data grows.

Make informed decisions about which algorithm to use based on efficiency.

Predict performance bottlenecks before they happen.

Search Operation Complexities

When analyzing search algorithms, it's helpful to consider different scenarios: the best-case, average-case, and worst-case performances.

1. Linear Search

Best-case: O(1) – If the element you’re searching for is the very first one.

Average-case: O(n) – On average, you’ll have to check about half the elements.

Worst-case: O(n) – You might have to check every single item, especially if the element isn’t there or is at the very end.

2. Binary Search

It only works with sorted data

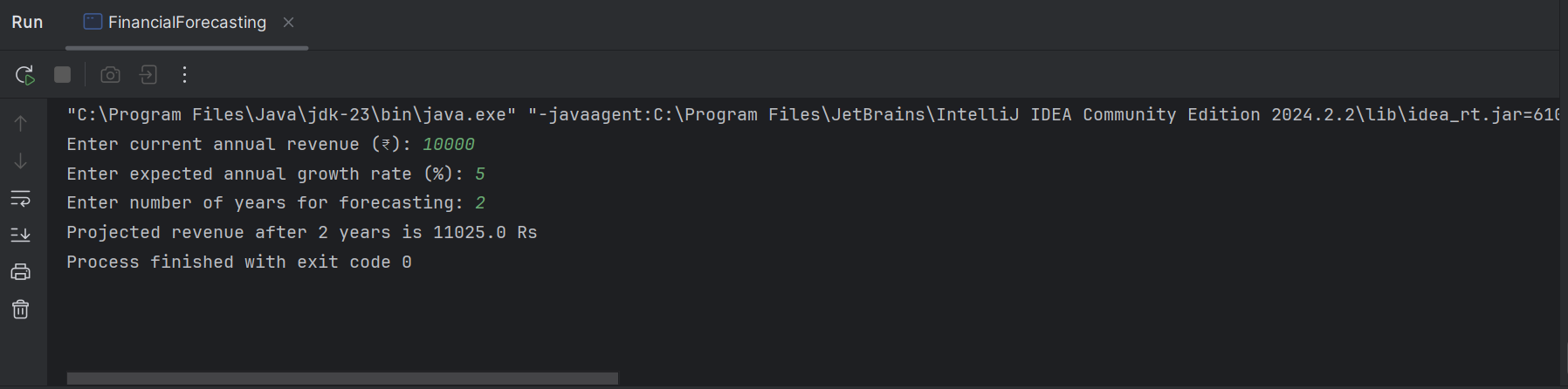
Best-case: O(1) – You get lucky, and the element is right in the middle.

Average-case: O(log n) – The search keeps halving the data until it finds the element.

Worst-case: O(log n) – Even in the worst case, the search only needs a few divisions to reach a conclusion.

**Q2 Financial Forecasting-**

import java.util.Scanner;  
  
public class FinancialForecasting {  
  
 public static double calculateFutureRevenue(double currentRevenue, double growthRate, int years) {  
 return currentRevenue \* Math.*pow*((1 + growthRate / 100), years);  
 }  
  
 public static void main(String[] args) {  
 Scanner input = new Scanner(System.*in*);  
  
 System.*out*.print("Enter current annual revenue (₹): ");  
 double revenue = input.nextDouble();  
  
 System.*out*.print("Enter expected annual growth rate (%): ");  
 double rate = input.nextDouble();  
  
 System.*out*.print("Enter number of years for forecasting: ");  
 int years = input.nextInt();  
  
 double forecast = *calculateFutureRevenue*(revenue, rate, years);  
  
 System.*out*.printf("Projected revenue after " + years + " years is " + forecast + " Rs");  
 }  
}



Recursion

Recursion is a technique where a function calls itself to solve smaller chunks of a larger problem. It's like solving a puzzle by breaking it down into easier parts, each of which looks a lot like the original.

**Key Parts of Recursion**

**Base Case**  
This is the simplest scenario where the function doesn't call itself. It acts as a stopping point to avoid an infinite loop.  
*Example:* In calculating future revenue, the base case is when years = 0, and the revenue remains the same.

**Recursive Case**  
This is where the function keeps calling itself with smaller inputs.  
*Example:* Future revenue is calculated by compounding the revenue each year:  
futureRevenue = currentRevenue \* (1 + rate)^years

**Why Recursion is Useful-**

**Breaks Down Problems**  
Recursion helps divide a problem into smaller, more manageable steps. Each function call handles a smaller piece of the puzzle.

**Cleaner Code**  
For some problems, recursion leads to shorter, more readable code—especially when dealing with structures like trees or graphs.

**Perfect for Certain Problems**  
Tasks like tree traversal, backtracking, or computing mathematical sequences (like Fibonacci or permutations) are naturally recursive.

**Things to Watch Out For-**

**Stack Overflow**  
Too many recursive calls can overload the system's call stack, especially if the base case is missing or unreachable.

**Performance Issues**  
Some recursive solutions repeat the same calculations many times. Using memoization (storing results) can help.

**Harder to Debug**  
It can be tricky to trace the flow in recursive programs, especially when there are multiple recursive calls involved.

**Performance of a Recursive Future Revenue Calculation with Memoization-**

**Time Complexity**  
Each unique value of years is computed once and stored.  
So, for n years, the time complexity is O(n).

**Space Complexity**  
Recursive calls go n levels deep: O(n) for the call stack.  
The memoization map stores n results: O(n).

**Better Alternatives**

**1. Iterative Method**

public static double calculateFutureRevenueIterative(double revenue, double rate, int years) {

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

revenue \*= (1 + rate / 100);

}

return revenue;

}

Time Complexity**:** O(n)  
Space Complexity**:** O(1)

**2. Using Formula**

public static double calculateFutureRevenueFormula(double revenue, double rate, int years) {

return revenue \* Math.pow(1 + rate / 100, years);

}

Time Complexity**:** O(1)  
Uses the compound interest formula and works well when exact mathematical computation is preferred.