Exercise -2

* Big O notation describes the **upper bound** of an algorithm's running time in terms of input size n. It helps compare algorithm efficiency by focusing on **growth rate,** not exact timings.
* In large-scale systems like e-commerce platforms, efficiency directly affects performance and user experience. Fast search operations improve response times and scalability.
* Best case – O(1) for both cases.
* Average case – O(n) for linear search and O(log n) for binary search.
* Worst case - O(n) for linear search and O(log n) for binary search.

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Phone", "Electronics"),

new Product(3, "Shoes", "Fashion"),

new Product(4, "Watch", "Accessories")

};

Product result1 = SearchOperations.linearSearch(products, "Watch");

if (result1 != null) {

System.out.println("Linear Search: Found -> " + result1);

} else {

System.out.println("Linear Search: Product not found.");

}

Product result2 = SearchOperations.binarySearch(products, "Watch");

if (result2 != null) {

System.out.println("Binary Search: Found -> " + result2);

} else {

System.out.println("Binary Search: Product not found.");

}

}

}

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;

}

public String toString() {

return "ProductID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

class SearchOperations {

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

long startTime = System.nanoTime();

for (Product product : products) {

if (product.productName.equalsIgnoreCase(targetName)) {

long endTime = System.currentTimeMillis();

System.out.println("Linear Search Time: " + (endTime - startTime) + " ns");

return product;

}

}

long endTime = System.nanoTime();

System.out.println("Linear Search Time: " + (endTime - startTime) + " ns");

return null;

}

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

long startTime = System.nanoTime();

java.util.Arrays.sort(products, (a, b) -> a.productName.compareToIgnoreCase(b.productName));

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

String midName = products[mid].productName;

if (midName.equalsIgnoreCase(targetName)) {

long endTime = System.nanoTime();

System.out.println("Binary Search Time: " + (endTime - startTime) + " ns");

return products[mid];

} else if (targetName.compareToIgnoreCase(midName) < 0) {

right = mid - 1;

} else {

left = mid + 1;

}

}

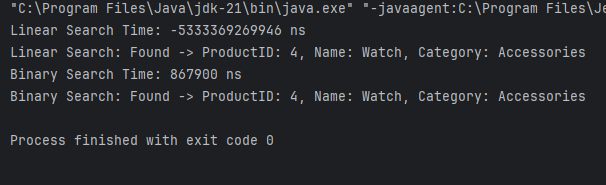
long endTime = System.nanoTime();

System.out.println("Binary Search Time: " + (endTime - startTime) + " ns");

return null;

}

}



* Binary search is best suited for search purposes because of time complexity O(log n) in the worst case.
* This helps in faster search process and good user experience.

Exercise – 7

* **Recursion** is a programming technique where a method calls itself to solve smaller instances of the same problem.
* In financial forecasting, future value often depends on previous values (like compound growth). Recursion can model this naturally.

public class FinancialForecasting {

public static double calculateFutureValue(double principal, double rate, int years) {

if (years == 0) {

return principal;

}

return calculateFutureValue(principal, rate, years - 1) \* (1 + rate);

}

public static void main(String[] args) {

double principal = 10000;

double rate = 0.05;

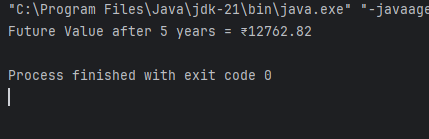
int years = 5;

double futureValue = calculateFutureValue(principal, rate, years);

System.out.printf("Future Value after %d years = ₹%.2f\n", years, futureValue);

}

}



* T(n) = T(n - 1) + O(1)
* Optimization: Replace Recursion with Iteration technique for large values of data.