Data Structures and Algorithms

Exercise 2: E-commerce Platform Search Function

Understand Asymptotic Notation:

Big O Notation:

- Big O notation describes the upper bound of an algorithm's running time in the worst-case scenario. It helps estimate how the algorithm's execution time or space grows with input size n.
- Common Big O notations:
  1. Constant time: O(1)
  2. Logarithmic time: O(log n)
  3. Linear time: O(n)
  4. Log-linear time: O(n log n)
  5. Quadratic time: O(n^2)
- It helps predict performance and allows comparison between algorithms.

Time complexity comparison between Linear Search and Binary Search

Algorithm	Best Case	Average Case	Worst Case
Linear Search	O(1) (first match)	O(n/2) ≈ O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

Setup and Implementation:

Code:

Product.java

```java
package algorithms.exercise2;

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 String getProductName() {
        return productName;
    }

    @Override
    public String toString() {
        return "Product{" +
                "productId=" + productId +
                ", productName='" + productName + '\'' +
                ", category='" + category + '\'' +
                '}';
    }
}
```

Client/linearSearch

```java
public static Product linearSearch(List<Product> products, String targetName) {
  for (Product product: products) {
    if (product.getProductName().equalsIgnoreCase(targetName)) {
      return product;
    }
  }
  return null;
}
```

Client/binarySearch

```java
  public static Product binarySearch(List<Product> products, String targetName) {
        int left = 0;
        int right = products.size() - 1;

        while (left <= right) {
            int mid = (left + right) / 2;
            Product midProduct = products.get(mid);
            int cmp = midProduct.getProductName().compareToIgnoreCase(targetName);

            if (cmp == 0) {
                return midProduct;
            } else if (cmp < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }
```

Client.java

```java
package algorithms.exercise2;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Client {
    // Linear search

    // Binary Search

    public static void main(String[] args) {
        List<Product> products = new ArrayList<>();
        products.add(new Product(101, "iPhone", "Mobile"));
        products.add(new Product(102, "MacBook", "Laptop"));
        products.add(new Product(103, "Galaxy Watch", "Watch"));
        products.add(new Product(104, "Dell XPS", "Laptop"));
        products.add(new Product(105, "OnePlus", "Mobile"));

        System.out.println("Product List:");
        for (Product product : products) {
            System.out.println(product);
        }

         // ---- Linear Search ----
        String searchName1 = "OnePlus";
        long startLinear = System.nanoTime();
        Product result1 = linearSearch(products, searchName1);
        long endLinear = System.nanoTime();
        long durationLinear = endLinear - startLinear;
        System.out.println("\nLinear Search Result for '" + searchName1 + "': " + result1);
        System.out.println("Time taken for Linear Search: " + durationLinear + " ns");

        // ---- Binary Search ----
        Collections.sort(products, (p1, p2) ->
            p1.getProductName().compareToIgnoreCase(p2.getProductName())
        );

        String searchName2 = "MacBook";
        long startBinary = System.nanoTime();
        Product result2 = binarySearch(products, searchName2);
        long endBinary = System.nanoTime();
        long durationBinary = endBinary - startBinary;
        System.out.println("\nBinary Search Result for '" + searchName2 + "': " + result2);
        System.out.println("Time taken for Binary Search: " + durationBinary + " ns");
    }
}
```

Output:

```
Product List:
Product{productId=101, productName='iPhone', category='Mobile'}
Product{productId=102, productName='MacBook', category='Laptop'}
Product{productId=103, productName='Galaxy Watch', category='Watch'}
Product{productId=104, productName='Dell XPS', category='Laptop'}
Product{productId=105, productName='OnePlus', category='Mobile'}

Linear Search Result for 'OnePlus': Product{productId=105, productName='OnePlus', category='Mobile'}
Time taken for Linear Search: 8900 ns

Binary Search Result for 'MacBook': Product{productId=102, productName='MacBook', category='Laptop'}
Time taken for Binary Search: 4700 ns
```

Analysis:

1. **Linear Search:**
    - Time complexity: O(n)
    - Doesn't require a sorted array (based on the item(s) we want to perform search operations).
    - Suitable for smaller and unsorted datasets.
2. **Binary Search:**
    - Time complexity: O(log n)
    - Required sorted array
    - Suitable for larger datasets.

Exercise 7: Financial Forecasting

Understand Recursive Algorithms

- Recursion is a programming technique where a method calls itself to solve a problem by breaking it down into smaller sub-problems.
- Here, the function calls itself with a reduced problem size.
- It provided a clear and concise implementation for problems with repeated patterns or steps.

Setup and Implementation

FinancialForecast.java

```java
package algorithms.exercise7;

public class FinancialForecast {

    public static double futureValue(double presentValue, double rate, int years) {
        if (years == 0) {  // Base case: no growth after 0 years
            return presentValue;
        }

        return (1 + rate) * futureValue(presentValue, rate, years - 1);
    }

    public static void main(String[] args) {
        double presentValue = 10000;
        double rate = 0.05;
        int years = 5;

        double futureVal = futureValue(presentValue, rate, years);
        System.out.printf("Future Value after %d years: $%.2f\n", years, futureVal);
    }
}
```

Result:

```
Present value: 10000.0
Rate of growth annually: 5.0%
Years: 5

Future Value after 5 years: $12762.82
```

Analysis

1. **Time Complexity:** O(n) where n is the number of years.
    - Each recursive call processes one year.
2. **Space Complexity:** O(n) due to the call stack.