**WEEK 2:DATA STRUCTURES AND ALGORITHMS**

**MANDATORY**

**Exercise 2: E-commerce Platform Search Function**

**Understanding Big O for Search**

Big O notation explains how fast or slow an algorithm performs as the number of products increases.

* Linear Search goes through each product one at a time until it finds a match or reaches the end.
* Binary Search works only on a sorted list and narrows down the search area by half each time.

**Best, Average, and Worst Case Scenarios**

**Linear Search:**

* **Best Case:** Product is the first in the list → O(1)
* **Average Case:** Product is somewhere in the middle → O(n/2)
* **Worst Case:** Product is at the end or not found → O(n)

**Binary Search:**

* **Best Case:** Product is in the middle → O(1)
* **Average/Worst Case:** Product is not found or requires several steps → O(log n)

**Which Search is Better?**

* Use Linear Search when:
  + The list is small
  + The list is unsorted
  + You want a quick and simple solution
* Use Binary Search when:
  + The list is large
  + The list is sorted
  + You want faster results for frequent searches

**Exercise 2: E-commerce Platform Search Function**

**SOLUTION:**

**Product.java**

package weekone;

class Product {

int id;

String name;

String category;

Product(int id, String name, String category) {

this.id = id;

this.name = name;

this.category = category;

}

}

**SearchFunction.java**

package weekone;

import java.util.Arrays;

public class SearchFunction {

public static int linearSearch(Product[] items, String target) {

for (int i = 0; i < items.length; i++) {

if (items[i].name.equalsIgnoreCase(target)) {

return i;

}

}

return -1;

}

public static int binarySearch(Product[] items, String target) {

int low = 0;

int high = items.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int cmp = items[mid].name.compareToIgnoreCase(target);

if (cmp == 0) return mid;

if (cmp < 0) low = mid + 1;

else high = mid - 1;

}

return -1;

}

public static void main(String[] args) {

Product[] items = {

new Product(101, "Laptop", "Electronics"),

new Product(102, "Mobile", "Electronics"),

new Product(103, "Shoes", "Footwear"),

new Product(104, "Watch", "Accessories"),

new Product(105, "Shirt", "Clothing")

};

int index1 = *linearSearch*(items, "Mobile");

if (index1 != -1) {

System.*out*.println("Found using Linear Search: " + items[index1].name);

} else {

System.*out*.println("Not found using Linear Search");

}

Arrays.*sort*(items, (a, b) -> a.name.compareToIgnoreCase(b.name));

int index2 = *binarySearch*(items, "Mobile");

if (index2 != -1) {

System.*out*.println("Found using Binary Search: " + items[index2].name);

} else {

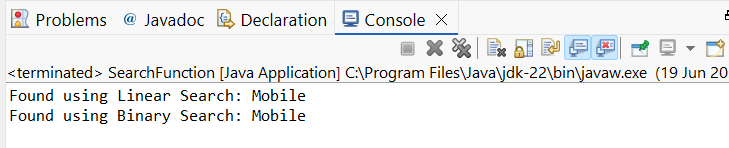
System.*out*.println("Not found using Binary Search");

}

}

}

**CONSOLE OUTPUT FOR EXERCISE 2:**



**MANDATORY**

**Exercise 7: Financial Forecasting**

**Understanding Recursive Algorithms**

Recursion is when a method calls itself to solve smaller instances of a larger problem. It continues doing so until it reaches a **base case**, which stops the recursion. Recursion is useful when a task repeats in a similar way multiple times.

In the code, the forecastValue method keeps calling itself to simulate each passing year of financial growth. For every year, it calculates the new value by multiplying the current amount by (1 + growthRate) and reduces the year count. Once the years reach 0, the recursion stops.

**Time Complexity**

The method runs **once for every year** until the number of years reaches zero, so the time complexity is **O(n)**, where n is the number of years. Each recursive call performs a small and constant amount of calculation.

**Optimization**

This recursive approach is clean and works well for a small or medium number of years. However, recursion can be less efficient for large values due to:

* Memory overhead from too many recursive calls
* Risk of stack overflow if the recursion goes too deep

In such cases, an iterative loop version is safer.

Since this function doesn’t repeat any calculations (unlike Fibonacci or tree problems), memoization isn't needed here.

**SOLUTION:**

package weekone;

public class FinancialForecast {

public static double forecastValue(double currentValue, double growthRate, int years) {

if (years == 0) {

return currentValue;

}

double nextValue = currentValue \* (1 + growthRate);

return *forecastValue*(nextValue, growthRate, years - 1);

}

public static void main(String[] args) {

double presentValue = 10000.0;

double annualGrowth = 0.08;

int forecastYears = 5;

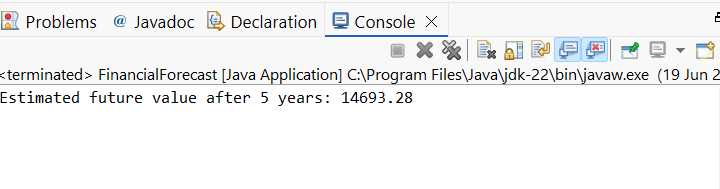
double futureValue = *forecastValue*(presentValue, annualGrowth, forecastYears);

System.*out*.printf("Estimated future value after %d years: %.2f", forecastYears, futureValue);

}

}

**CONSOLE OUTPUT FOR EXERCISE 7:**

****

**HANDS ON**

**Exercise 1: Inventory Management System**

SOLUTION:

package weekone;

import java.util.HashMap;

class Product {

int productId;

String productName;

int quantity;

double price;

Product(int productId, String productName, int quantity, double price) {

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

}

public class InventoryManagementSystem {

static HashMap<Integer, Product> *inventory* = new HashMap<>();

public static void addProduct(Product p) {

*inventory*.put(p.productId, p);

}

public static void updateProduct(int id, int quantity, double price) {

if (*inventory*.containsKey(id)) {

Product p = *inventory*.get(id);

p.quantity = quantity;

p.price = price;

}

}

public static void deleteProduct(int id) {

*inventory*.remove(id);

}

public static void main(String[] args) {

*addProduct*(new Product(101, "Monitor", 20, 7999.50));

*addProduct*(new Product(102, "Keyboard", 50, 999.99));

*updateProduct*(101, 25, 7499.00);

*deleteProduct*(102);

for (Product p : *inventory*.values()) {

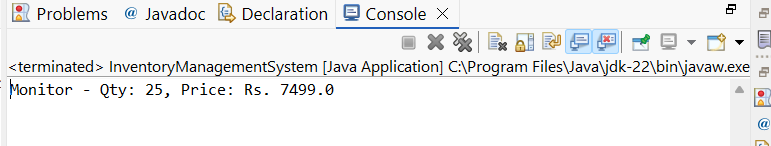
System.*out*.println(p.productName + " - Qty: " + p.quantity + ", Price: Rs. " + p.price);

}

}

}

**CONSOLE OUTPUT FOR EXERCISE 1:**

****

**Exercise 3: Sorting Customer Orders**

SOLUTION:

package weekone;

class Order {

int orderId;

String customerName;

double totalPrice;

Order(int orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

}

public class CustomerOrderSorting {

public static void bubbleSort(Order[] orders) {

for (int i = 0; i < orders.length - 1; i++) {

for (int j = 0; j < orders.length - i - 1; j++) {

if (orders[j].totalPrice > orders[j + 1].totalPrice) {

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

public static void quickSort(Order[] orders, int low, int high) {

if (low < high) {

int pi = *partition*(orders, low, high);

*quickSort*(orders, low, pi - 1);

*quickSort*(orders, pi + 1, high);

}

}

private static int partition(Order[] orders, int low, int high) {

double pivot = orders[high].totalPrice;

int i = low - 1;

for (int j = low; j < high; j++) {

if (orders[j].totalPrice <= pivot) {

i++;

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

Order temp = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp;

return i + 1;

}

public static void main(String[] args) {

Order[] orders = {

new Order(101, "Subhiksha", 5000),

new Order(102, "Yega", 3500),

new Order(103, "Aisu", 8200),

new Order(104, "Sam", 2000)

};

*bubbleSort*(orders);

System.*out*.println("Sorted using Bubble Sort:");

for (Order o : orders) {

System.*out*.println(o.customerName + " - Rs. " + o.totalPrice);

}

Order[] orders2 = {

new Order(101, "Subhiksha", 5000),

new Order(102, "Yega", 3500),

new Order(103, "Aisu", 8200),

new Order(104, "Sam", 2000)

};

*quickSort*(orders2, 0, orders2.length - 1);

System.*out*.println("Sorted using Quick Sort:");

for (Order o : orders2) {

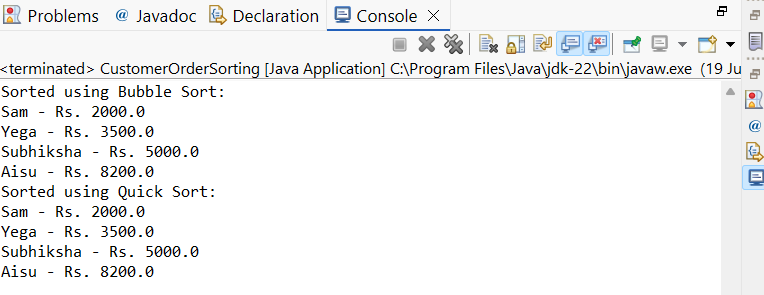
System.*out*.println(o.customerName + " - Rs. " + o.totalPrice);

}

}

}

**CONSOLE OUTPUT FOR EXERCISE 3:**

****

**Exercise 4: Employee Management System**

**SOLUTION:**

package weekone;

class Employee {

int employeeId;

String name;

String position;

double salary;

Employee(int employeeId, String name, String position, double salary) {

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

}

public class EmployeeManagementSystem {

static Employee[] *employees* = new Employee[5];

static int *count* = 0;

public static void addEmployee(Employee e) {

if (*count* < *employees*.length) {

*employees*[*count*++] = e;

}

}

public static Employee searchEmployee(int id) {

for (int i = 0; i < *count*; i++) {

if (*employees*[i].employeeId == id) {

return *employees*[i];

}

}

return null;

}

public static void deleteEmployee(int id) {

for (int i = 0; i < *count*; i++) {

if (*employees*[i].employeeId == id) {

for (int j = i; j < *count* - 1; j++) {

*employees*[j] = *employees*[j + 1];

}

*employees*[--*count*] = null;

break;

}

}

}

public static void displayEmployees() {

for (int i = 0; i < *count*; i++) {

System.*out*.println(*employees*[i].name + " - " + *employees*[i].position);

}

}

public static void main(String[] args) {

*addEmployee*(new Employee(1, "Subhiksha", "Manager", 50000));

*addEmployee*(new Employee(2, "Aisu", "Clerk", 25000));

*displayEmployees*();

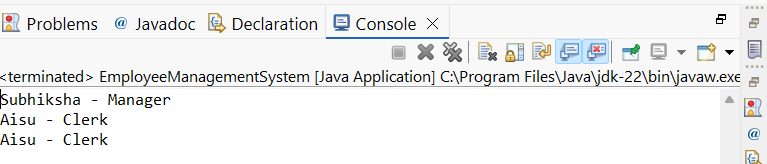
*deleteEmployee*(1);

*displayEmployees*();

}

}

**CONSOLE OUTPUT FOR EXERCISE 4:**

****

**Exercise 5: Task Management System**

**SOLUTION:**

package weekone;

class Task {

int taskId;

String taskName;

String status;

Task next;

Task(int taskId, String taskName, String status) {

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

this.next = null;

}

}

class TaskManager {

Task head;

public void addTask(Task task) {

if (head == null) {

head = task;

} else {

Task temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = task;

}

}

public Task searchTask(int id) {

Task temp = head;

while (temp != null) {

if (temp.taskId == id) {

return temp;

}

temp = temp.next;

}

return null;

}

public void deleteTask(int id) {

if (head == null) return;

if (head.taskId == id) {

head = head.next;

return;

}

Task temp = head;

while (temp.next != null) {

if (temp.next.taskId == id) {

temp.next = temp.next.next;

return;

}

temp = temp.next;

}

}

public void displayTasks() {

Task temp = head;

while (temp != null) {

System.*out*.println(temp.taskName + " - " + temp.status);

temp = temp.next;

}

}

public static void main(String[] args) {

TaskManager manager = new TaskManager();

manager.addTask(new Task(1, "Design", "Pending"));

manager.addTask(new Task(2, "Development", "In Progress"));

manager.displayTasks();

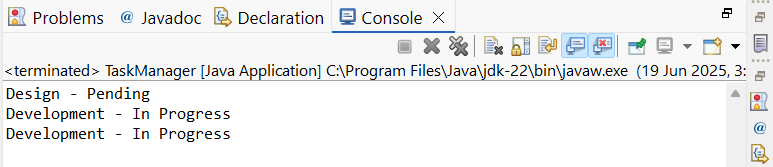
manager.deleteTask(1);

manager.displayTasks();

}

}

**CONSOLE OUTPUT FOR EXERCISE 5:**

****

**Exercise 6: Library Management System**

**SOLUTION:**

package weekone;

class Book {

int bookId;

String title;

String author;

Book(int bookId, String title, String author) {

this.bookId = bookId;

this.title = title;

this.author = author;

}

}

public class LibrarySearch {

public static Book linearSearch(Book[] books, String title) {

for (Book b : books) {

if (b.title.equalsIgnoreCase(title)) {

return b;

}

}

return null;

}

public static Book binarySearch(Book[] books, String title) {

int low = 0, high = books.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int cmp = books[mid].title.compareToIgnoreCase(title);

if (cmp == 0) return books[mid];

else if (cmp < 0) low = mid + 1;

else high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Book[] books = {

new Book(1, "Algorithms", "Subhiksha"),

new Book(2, "Data Structures", "Aisu"),

new Book(3, "Operating Systems", "Yega")

};

Book result1 = *linearSearch*(books, "Data Structures");

System.*out*.println("Linear Search Found: " + result1.title);

Book[] sortedBooks = {

new Book(1, "Algorithms", "Subhiksha"),

new Book(2, "Data Structures", "Aisu"),

new Book(3, "Operating Systems", "Yega")

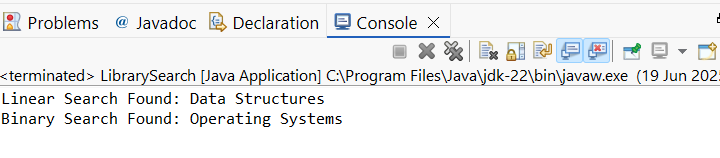
};

Book result2 = *binarySearch*(sortedBooks, "Operating Systems");

System.*out*.println("Binary Search Found: " + result2.title);

}

**CONSOLE OUTPUT FOR EXERCISE 6:**

****