HANDSON EXERCISES - WEEK 1

Skill : Data Structures and Algorithms

Exercise 2: E-commerce Platform Search Function

Scenario:

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

Understand Asymptotic Notation:

Big O describes how fast or slow an algorithm is as the size of the input grows. It helps us compare which code runs faster or slower in the worst case.

- Best case (Ω) - fastest possible case in which the item we are searching for is found at the begininng itself
- Average case (θ) - general case when the item is somewhere in the middle
- Worst case (O) - Slowest case when the item is at the end or else not present at all.

CODE :

Product.java :

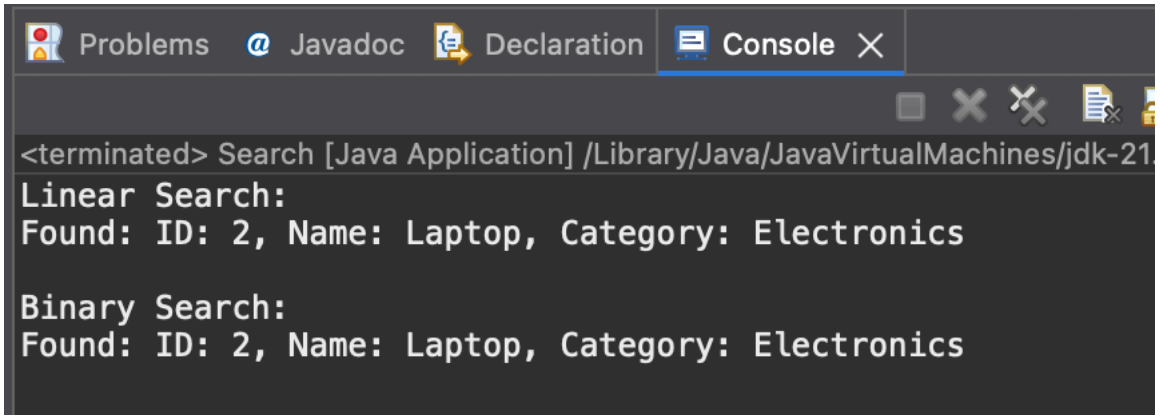
```
package com.example.Searching;

public 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;
	}

	@Override
	public String toString() {
		return "ID: " + productId + ", Name: " + productName + ", Category: " +
	category;
	}
}
```

Search.java :

```
package com.example.Searching;

import java.util.Arrays;
import java.util.Comparator;
```

```
public class Search {
	public static int linearSearch(Product[] products, String key) {
		for (int i = 0; i < products.length; i++) {
			if (products[i].productName.equalsIgnoreCase(key)) {


				return i;
			}
		}
		return -1;
	}

	public static int binarySearch(Product[] products, String key) {
		int left = 0, right = products.length - 1;
		while (left <= right) {
			int mid = (left + right) / 2;
			int cmp = key.compareToIgnoreCase(products[mid].productName);
			if (cmp == 0) return mid;
			else if (cmp < 0) right = mid - 1;
			else left = mid + 1;
		}
		return -1;
	}

	public static void main(String[] args) {
		Product[] products = {
			new Product(1, "Shoes", "Footwear"),
			new Product(2, "Laptop", "Electronics"),
			new Product(3, "T-shirt", "Clothing"),
			new Product(4, "Phone", "Electronics"),
			new Product(5, "Watch", "Accessories")
		};

		String searchKey = "Laptop";

		int linearIndex = linearSearch(products, searchKey);
		System.out.println("Linear Search:");
		if (linearIndex != -1) {
			System.out.println("Found: " + products[linearIndex]);
		} else {
			System.out.println("Product not found.");
		}

		Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));
		int binaryIndex = binarySearch(products, searchKey);
		System.out.println("\nBinary Search:");
		if (binaryIndex != -1) {
			System.out.println("Found: " + products[binaryIndex]);
		} else {
			System.out.println("Product not found.");
		}
	}
}
```

Output :



Time Complexity Analysis:

- Linear Search :
  - Best Case - O(1)
  - Average / Worst Case - O(n)
- Binary Search :
  - Best case - O(1)
  - Average / Worst Case - O(log n)

The main difference is that binary search requires sorting , where as linear search searches through all linearly.

Best Approach for our problem :

- Linear Search is suitable for a small list or when there are few no. of products.
- Binary Search is suitable for frequent searches on a large list.

Exercise 7: Financial Forecasting

Scenario:

You are developing a financial forecasting tool that predicts future values based on past data.

Understanding Recursive Algorithms:

Recursion is when a method calls **itself** to solve a smaller version of the same problem.

But why use Recursion :?
It simplifies problems like :

- Factorials
- Fibonacci
- Tree traversals

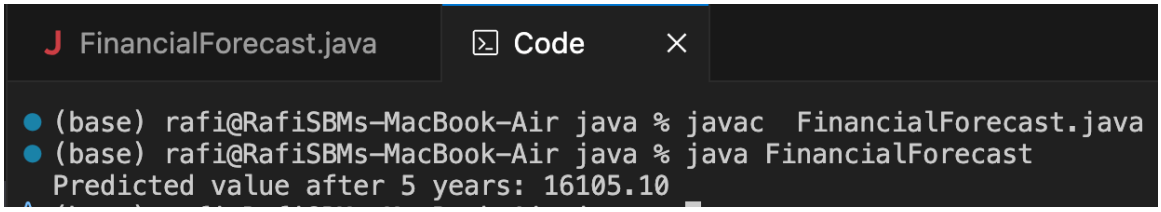
CODE

FinancialForecast.java :

```
public class FinancialForecast {
```

```
    public static double predictValue(double initialValue, double rate, int years) {
        if (years == 0) return initialValue;
        return (1 + rate) * predictValue(initialValue, rate, years - 1);
    }

    public static void main(String[] args) {
        double initialValue = 10000;
        double annualGrowthRate = 0.1;
        int years = 5;

        double futureValue = predictValue(initialValue, annualGrowthRate, years);

        System.out.printf("Predicted value after %d years: %.2f\n", years, futureValue);
    }
}
```

Output :



Time Complexity Analysis:

- T(n) = T(n-1) + O(1) --> O(n)

Linear time , but uses call stack memory

Optimizing :

Here the need of optimization is , cause recursion can:

- consume more memory -- stack overflow
- be slower than iterative solutions

can be optimized if we use iteration method , cause recursion takes O(n) for both space and time complexities , whereas iteration takes O(n) for time complexity and O(1) for space complexity ,leading to optimal speed.

Exercise 3: Sorting Customer Orders

Scenario:

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

Understand Sorting Algorithms:

- **Bubble Sort -** It keeps swapping nearby numbers if they're in the wrong order , again and again.
- **Insertion Sort -** It puts each number in the right place one by one like sorting plyaing cards

- **Quick Sort -** It picks a number , putts smaller ones on left , bigger on right,and repeats this.
- **Merge Sort -** It splits the list into halves, sorts them , and then joins them back together

CODE

Order.java :

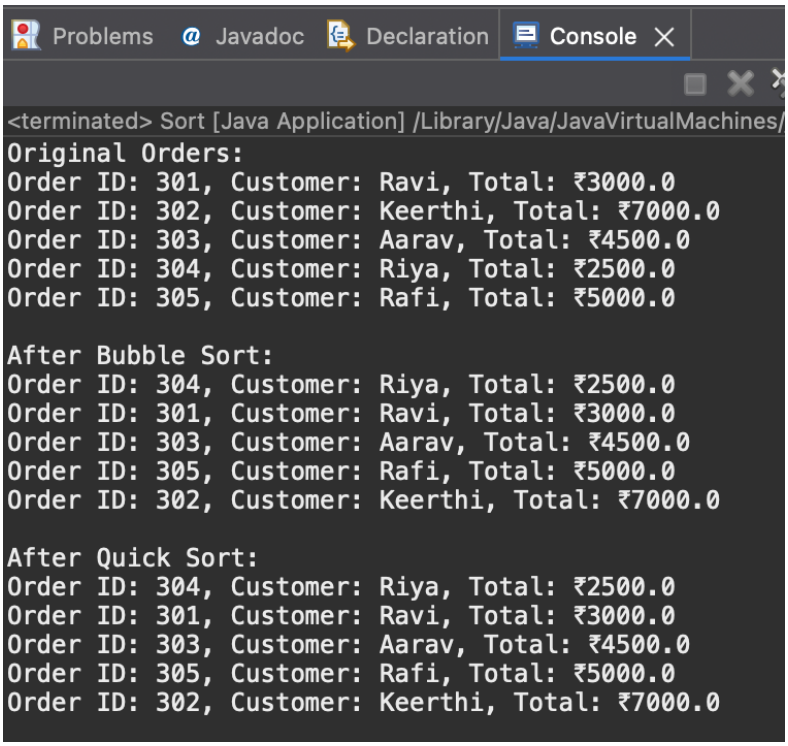
```
package com.example.Sorting;

public class Order {
	int orderId;
	String customerName;
	double totalPrice;

	public Order(int orderId, String customerName, double totalPrice) {
		this.orderId = orderId;
		this.customerName = customerName;
		this.totalPrice = totalPrice;
	}

	@Override
	public String toString() {
		return "Order ID: " + orderId + ",Customer: " + customerName + ", Total: ₹" + totalPrice;
	}
}
```

Sort.java :

```
package com.example.Sorting;

public class Sort {

	public static void bubbleSort(Order[] orders) {
		int n = orders.length;
		for (int i = 0; i < n - 1; i++) {
			for (int j = 0; j < n - 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);
		}
	}
```

```
    public 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 printOrders(Order[] orders) {
        for (Order o : orders) {
            System.out.println(o);
        }
    }

    public static void main(String[] args) {
        Order[] orders = {
            new Order(301, "Ravi", 3000),
            new Order(302, "Keerthi", 7000),
            new Order(303, "Aarav", 4500),
            new Order(304, "Riya", 2500),
            new Order(305, "Rafi", 5000)
        };

        Order[] bubbleSorted = orders.clone();
        Order[] quickSorted = orders.clone();

        System.out.println("Original Orders:");
        printOrders(orders);

        bubbleSort(bubbleSorted);
        System.out.println("\nAfter Bubble Sort:");
        printOrders(bubbleSorted);

        quickSort(quickSorted, 0, quickSorted.length - 1);
        System.out.println("\nAfter Quick Sort:");
        printOrders(quickSorted);
    }
}
```

Output :



Time Complexity Analysis:

- **Bubble Sort -** Avg O(n^2) , its stable and has a sapce complexity of O(1)
- **Quick Sort -** Avg O(n log n) , it is not stable and has a space complexity of O(log n)

Why Quick Sort is Prefered :

- Much faster for larger inputs , used internally by Java's Arrays.sort() for primitive types
- More efficient in real world cases

Exercise 4: Employee Management System

Scenario:

You are developing an employee management system for a company. Efficiently managing employee records is crucial.

Understanding Array Representation:

An array is a collection of elements stored at contiguous memory locations.

ex : int [] arr = {1,2,3} in memory looks looks like :

[1] [2] [3]

0 1 2 <-- Indexes

Advantages of Arrays :

- Can be able to access any element in O(1) time using index
- We can loop thought it easily , uses a continuous block of memory

CODE

Employee.java :

```
package com.example.Arrays;

public class Employee {
	int employeeId;
	String name;
	String position;
	double salary;

	public Employee(int employeeId, String name, String position, double salary) {
		this.employeeId = employeeId;
		this.name = name;
		this.position = position;
		this.salary = salary;
	}

	@Override
	public String toString() {
		return "ID: " + employeeId + ", Name: " + name + ", Position: " + position
	+ ", Salary: " + salary;
	}
}
```

EmployeeSystem.java :

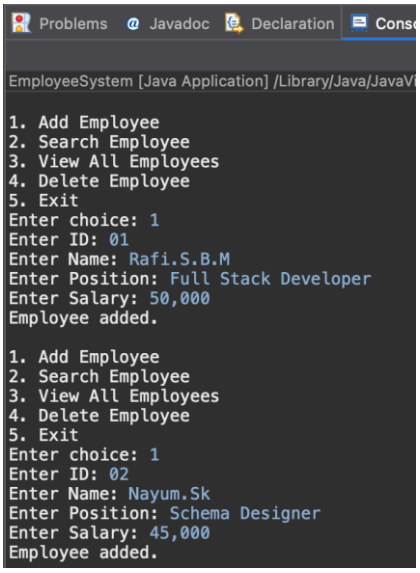
```
package com.example.Arrays;

import java.util.Scanner;

public class EmployeeSystem {
	static final int MAX_EMPLOYEES = 100;
	static Employee[] employees = new Employee[MAX_EMPLOYEES];
	static int count = 0;

	public static void addEmployee(Employee e) {
		if (count < MAX_EMPLOYEES) {
			employees[count++] = e;
			System.out.println("Employee added.");
		} else {
			System.out.println("Employee limit reached.");
		}
	}

	public static void searchEmployee(int id) {
		for (int i = 0; i < count; i++) {
			if (employees[i].employeeId == id) {
				System.out.println("Found: " + employees[i]);
				return;
			}
		}
		System.out.println("Employee not found.");
	}
```

```
public static void traverseEmployees() {
        if (count == 0) {
                System.out.println("No employees to display.");
                return;
        }
        for (int i = 0; i < count; i++) {
                System.out.println(employees[i]);
        }
}

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]; // Shift left
                        }
                        employees[--count] = null;
                        System.out.println("Employee deleted.");
                        return;
                }
        }
        System.out.println("Employee not found.");
}

public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int choice;
        do {
                System.out.println("\n1. Add Employee\n2. Search Employee\n3.
        View All Employees\n4. Delete Employee\n5. Exit");
                System.out.print("Enter choice: ");
                choice = sc.nextInt();

                switch (choice) {
                        case 1:
                                System.out.print("Enter ID: ");
                                int id = sc.nextInt();
                                sc.nextLine();
                                System.out.print("Enter Name: ");
                                String name = sc.nextLine();
                                System.out.print("Enter Position: ");
                                String pos = sc.nextLine();
                                System.out.print("Enter Salary: ");
                                double sal = sc.nextDouble();
                                addEmployee(new Employee(id, name, pos, sal));
                                break;

                        case 2:
                                System.out.print("Enter ID to search: ");
                                int sid = sc.nextInt();
                                searchEmployee(sid);
                                break;
```

```
                case 3:
                    traverseEmployees();
                    break;

                case 4:
                    System.out.print("Enter ID to delete: ");
                    int did = sc.nextInt();
                    deleteEmployee(did);
                    break;

                case 5:
                    System.out.println("Exiting...");
                    break;

                default:
                    System.out.println("Invalid choice!");
            }

        } while (choice != 5);
        sc.close();
    }
}
```

Output :

Adding an employee :



Searching and Viewing employees :

```
1. Add Employee
2. Search Employee
3. View All Employees
4. Delete Employee
5. Exit
Enter choice: 2
Enter ID to search: 02
Found: ID: 2, Name: Nayum.Sk, Position: Schema Designer, Salary: 45000.0

1. Add Employee
2. Search Employee
3. View All Employees
4. Delete Employee
5. Exit
Enter choice: 3
ID: 1, Name: Rafi.S.B.M, Position: Full Stack Developer, Salary: 50000.0
ID: 2, Name: Nayum.Sk, Position: Schema Designer, Salary: 45000.0
```

Deleting an Employee :

```
1. Add Employee
2. Search Employee
3. View All Employees
4. Delete Employee
5. Exit
Enter choice: 4
Enter ID to delete: 02
Employee deleted.

1. Add Employee
2. Search Employee
3. View All Employees
4. Delete Employee
5. Exit
Enter choice: 3
ID: 1, Name: Rafi.S.B.M, Position: Full Stack Developer, Salary: 50000.0
```

Time Complexity Analysis:

- Adding an element will take O(1) , Search and traversal,deleting of employees - O(n) ,

Limitations of using Arrays:

- **Fixed Size -** The size cannot increase beyond size (MAX_EMPLOYEES)
- Can't esaily insert between elements
- **Costly delete operations ,** better alternations are ArrayList , HashMap for more flexibility.

Exercise 5: Task Management System

Scenario:

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

Understanding Linked Lists:

- **Single Linked List :** Each item points to the next one only.
- **Doubly Linked List :** Each item points to both next and previous ones.

CODE :

Task.java

```
package com.example.Linked_List;

public class Task {
        int taskId;
        String taskName;
        String status;
        Task next;

        public Task(int taskId, String taskName, String status) {
                this.taskId = taskId;
                this.taskName = taskName;
                this.status = status;
                this.next = null;
        }
```

```
    @Override
    public String toString() {
        return "ID: " + taskId + ", Name: " + taskName + ", Status: " + status;
    }
}
```

TaskLinkedList.java

```
package com.example.Linked_List;

public class TaskLinkedList {
    Task head;

    public void addTask(int id, String name, String status) {
        Task newTask = new Task(id, name, status);
        if (head == null) {
            head = newTask;
        } else {
            Task current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newTask;
        }
        System.out.println("Task added.");
    }

    public void searchTask(int id) {
        Task current = head;
        while (current != null) {
            if (current.taskId == id) {
                System.out.println("Found: " + current);
                return;
            }
            current = current.next;
        }
        System.out.println("Task not found.");
    }

    public void traverseTasks() {
        if (head == null) {
            System.out.println("No tasks found.");
            return;
        }
        Task current = head;
        while (current != null) {
            System.out.println(current);
            current = current.next;
        }
    }
```

```
    public void deleteTask(int id) {
        if (head == null) {
            System.out.println("No tasks to delete.");
            return;
        }

        if (head.taskId == id) {
            head = head.next;
            System.out.println("Task deleted.");
            return;
        }

        Task current = head;
        while (current.next != null) {
            if (current.next.taskId == id) {
                current.next = current.next.next;
                System.out.println("Task deleted.");
                return;
            }
            current = current.next;
        }
        System.out.println("Task not found.");
    }
}
```

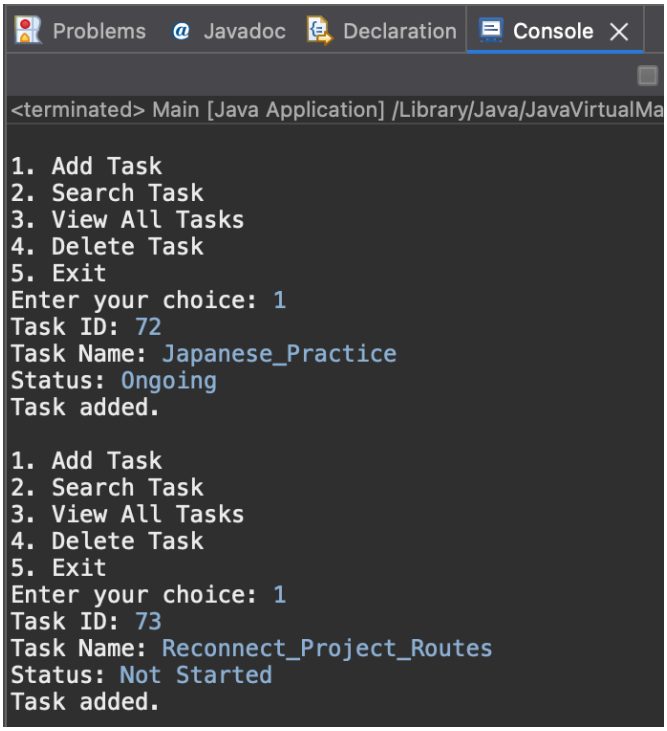
Main.java :

```
package com.example.Linked_List;

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        TaskLinkedList taskList = new TaskLinkedList();
        Scanner sc = new Scanner(System.in);
        int choice;

        do {
            System.out.println("\n1. Add Task\n2. Search Task\n3. View All
        Tasks\n4. Delete Task\n5. Exit");
            System.out.print("Enter your choice: ");
            choice = sc.nextInt();

            switch (choice) {
                case 1:
                    System.out.print("Task ID: ");
                    int id = sc.nextInt();
                    sc.nextLine();
                    System.out.print("Task Name: ");
                    String name = sc.nextLine();
                    System.out.print("Status: ");
                    String status = sc.nextLine();
                    taskList.addTask(id, name, status);
                    break;
```

```
                case 2:
                    System.out.print("Enter Task ID to search: ");
                    int sid = sc.nextInt();
                    taskList.searchTask(sid);
                    break;

                case 3:
                    taskList.traverseTasks();
                    break;

                case 4:
                    System.out.print("Enter Task ID to delete: ");
                    int did = sc.nextInt();
                    taskList.deleteTask(did);
                    break;

                case 5:
                    System.out.println("Exiting...");
                    break;

                default:
                    System.out.println("Invalid choice!");
                }
            } while (choice != 5);
            sc.close();
        }
}
```

Output :

Adding Tasks :



Viewing Tasks :

```
1. Add Task
2. Search Task
3. View All Tasks
4. Delete Task
5. Exit
Enter your choice: 3
ID: 72, Name: Japanese_Practice, Status: Ongoing
ID: 73, Name: Reconnect_Project_Routes, Status: Not Started
```

Searching a task :

```
1. Add Task
2. Search Task
3. View All Tasks
4. Delete Task
5. Exit
Enter your choice: 2
Enter Task ID to search: 73
Found: ID: 73, Name: Reconnect_Project_Routes, Status: Not Started
```

Deleting a task :

```
1. Add Task
2. Search Task
3. View All Tasks
4. Delete Task
5. Exit
Enter your choice: 4
Enter Task ID to delete: 73
Task deleted.

1. Add Task
2. Search Task
3. View All Tasks
4. Delete Task
5. Exit
Enter your choice: 3
ID: 72, Name: Japanese_Practice, Status: Ongoing
```

Time Complexity Analysis:

- Adding - at end , searching , traversal , deleting tasks - O(n)

Advantages if Linked List over Array :

- It can have dynamic Size which grows as per the need
- Easy Adjustment of pointers leading to felxible inseritons and deletions
- Memory is Scattered and not continuous

Exercise 6: Library Management System

Scenario:

You are developing a library management system where users can search for books by title or author.

Understanding Search Algorithms:

- **Linear Search -** Go one by one through the list to find the item, works on unsorted data.
- **Binary Search -** Keep cutting the sorted list in half to find the item faster, only works on sorted data.

CODE

Book.java :

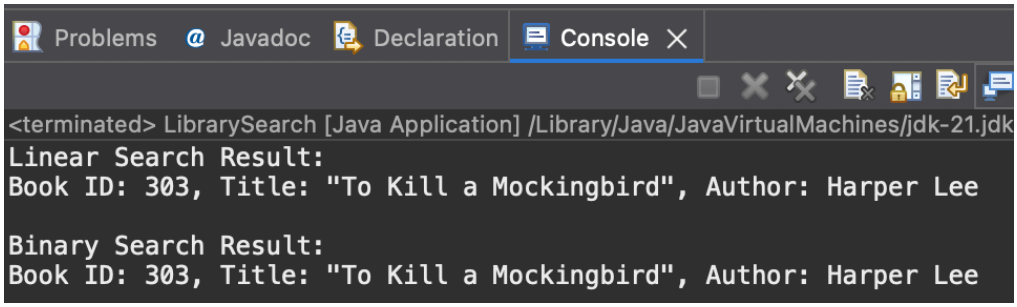
```
package com.example.Search;

public class Book {
	int bookId;
	String title;
	String author;

	public Book(int bookId, String title, String author) {
		this.bookId = bookId;
		this.title = title;
		this.author = author;
	}

	@Override
	public String toString() {
		return "Book ID: " + bookId + ", Title: \"" + title + "\", Author: " + author;
	}
}
```

LibrarySearch.java :

```
package com.example.Search;

import java.util.Arrays;
import java.util.Comparator;

public class LibrarySearch {

// Linear Search by title
	public static int linearSearch(Book[] books, String title) {
		for (int i = 0; i < books.length; i++) {
			if (books[i].title.equalsIgnoreCase(title)) {
				return i;
			}
		}
		return -1;
	}
```

```
    public static int binarySearch(Book[] books, String title) {
        int left = 0, right = books.length - 1;

        while (left <= right) {
            int mid = (left + right) / 2;
            int cmp = title.compareToIgnoreCase(books[mid].title);
            if (cmp == 0) return mid;
            else if (cmp < 0) right = mid - 1;
            else left = mid + 1;
        }

        return -1;
    }

    public static void printBooks(Book[] books) {
        for (Book b : books) {
            System.out.println(b);
        }
    }

    public static void main(String[] args) {
        Book[] books = {
            new Book(301, "The Alchemist", "Paulo Coelho"),
            new Book(302, "1984", "George Orwell"),
            new Book(303, "To Kill a Mockingbird", "Harper Lee"),
            new Book(304, "Atomic Habits", "James Clear"),
            new Book(305, "The Great Gatsby", "F. Scott Fitzgerald")
        };

        String searchTitle = "To Kill a Mockingbird";

        int linearIndex = linearSearch(books, searchTitle);
        System.out.println("Linear Search result:");
        if (linearIndex != -1) {
            System.out.println(books[linearIndex]);
        } else {
            System.out.println("Book not found.");
        }

        Arrays.sort(books, Comparator.comparing(b -> b.title.toLowerCase()));
        int binaryIndex = binarySearch(books, searchTitle);
        System.out.println("\nBinary Search Result:");
        if (binaryIndex != -1) {
            System.out.println(books[binaryIndex]);
        } else {
            System.out.println("Book not found.");
        }
    }
}
```

Output :



Time Complexity Analysis:

- **Best case** - Both Linear Search , Binary Search has O(1)
- **Avg/Worst case**
  - Liner Search O(n);
  - Binary Search O(log n);

When to use which one ?

- If data is unsorted then go with Linear Search , if its sorted then go with Binary Search.
- Use Binary Search (with sorting beforehand) if there are large data sets and performance is needed

Exercise 1: Inventory Management System

Scenario:

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

Why are Data Structures and Algorithms important ?

- In a large warehouse , there might be having thousands of items.
- If we don't use proper data structure , searching or updating a product can become very slow.
- So , to make the system fast and efficient, we need to pick the right data structure.

Suitable Data Structures for this problem are :

i. **ArrayList** - Which is simple and keeps insertion order ; is best if we mostly add/display products.

ii. **HashMap** - Fast O(1) access via key ; best for quick searching,updating,deleting using product ID which is the best option for this problem.

CODE

product.java :

```
package com.example.inventory;

public class Product {
        int productId;
        String productName;
        int quantity;
        double price;
```

```
    public Product(int productId, String productName, int quantity, double price) {
        this.productId = productId;
        this.productName = productName;
        this.quantity = quantity;
        this.price = price;
    }

    @Override
    public String toString() {
        return "ID: " + productId + ", Name: " + productName + ", Quantity: " +
quantity + ", Price: ₹" + price;
    }
}
```

InventoryManager.java :

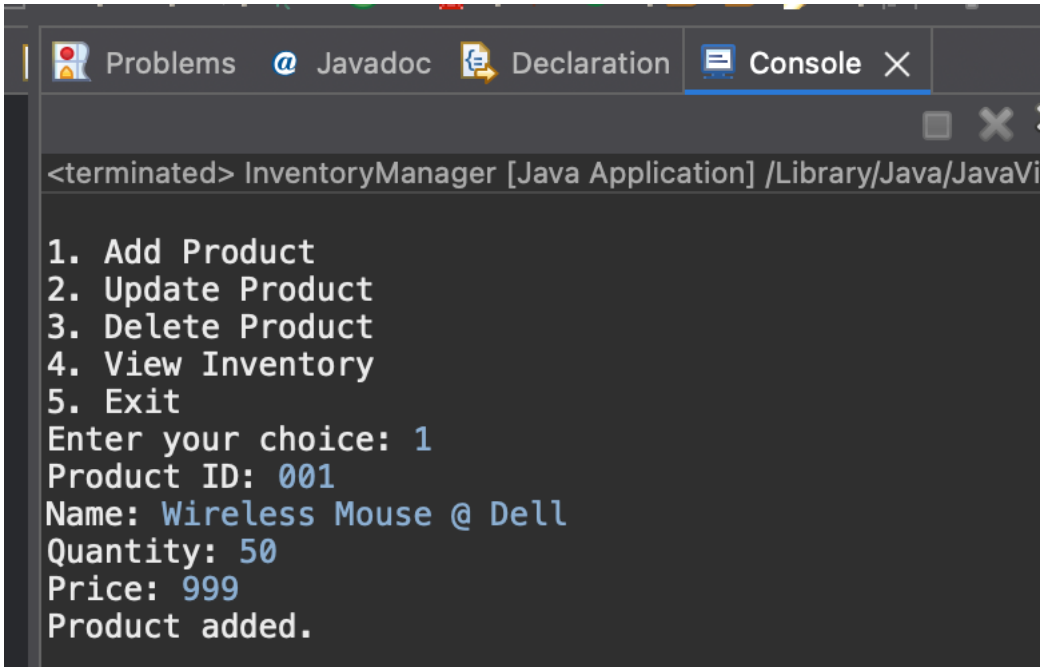
```
package com.example.inventory;

import java.util.HashMap;
import java.util.Scanner;

public class InventoryManager {
    static HashMap<Integer, Product> inventory = new HashMap<>();

    public static void addProduct(int id, String name, int quantity, double price) {
        if (inventory.containsKey(id)) {
            System.out.println("Product already exists with ID: " + id);
        } else {
            Product p = new Product(id, name, quantity, price);
            inventory.put(id, p);
            System.out.println("Product added.");
        }
    }

    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;
            System.out.println("Product updated.");
        } else {
            System.out.println("Product not found!");
        }
    }

    public static void deleteProduct(int id) {
        if (inventory.containsKey(id)) {
            inventory.remove(id);
            System.out.println("Product deleted.");
        } else {
            System.out.println("Product not found!");
        }
    }
```

```
public static void viewInventory() {
        if (inventory.isEmpty()) {
                System.out.println("Inventory is empty.");
        } else {
                for (Product p : inventory.values()) {
                        System.out.println(p);
                }
        }
}

public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int choice;


        do {
                System.out.println("\n1. Add Product\n2. Update Product\n3.
        Delete Product\n4. View Inventory\n5. Exit");
                System.out.print("Enter your choice: ");
                choice = sc.nextInt();

                switch (choice) {
                        case 1:
                                System.out.print("Product ID: ");
                                int id = sc.nextInt();
                                sc.nextLine();
                                System.out.print("Name: ");
                                String name = sc.nextLine();
                                System.out.print("Quantity: ");
                                int qty = sc.nextInt();
                                System.out.print("Price: ");
                                double price = sc.nextDouble();
                                addProduct(id, name, qty, price);
                                break;

                        case 2:
                                System.out.print("Product ID to update: ");
                                int uid = sc.nextInt();
                                System.out.print("New Quantity: ");
                                int uq = sc.nextInt();
                                System.out.print("New Price: ");
                                double up = sc.nextDouble();
                                updateProduct(uid, uq, up);
                                break;

                        case 3:
                                System.out.print("Product ID to delete: ");
                                int did = sc.nextInt();
                                deleteProduct(did);
                                break;

                        case 4:
                                viewInventory();
                                break;
```

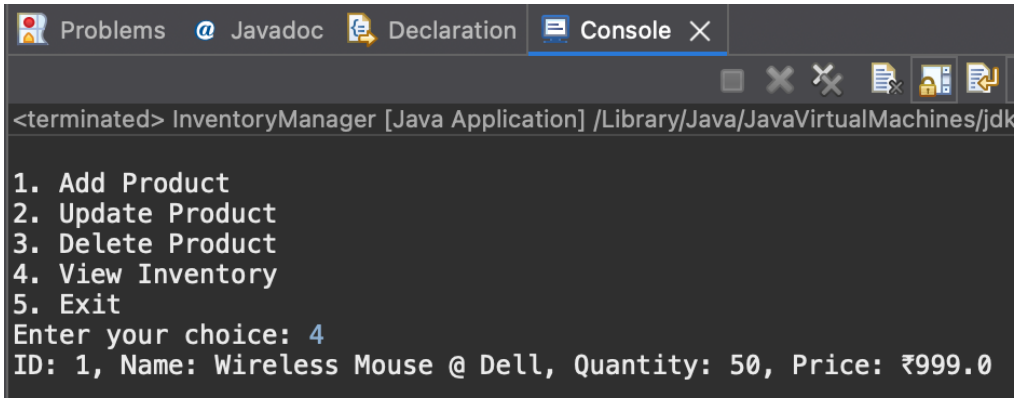
```
                case 5:
                    System.out.println("Exiting...");
                    break;

                default:
                    System.out.println("Invalid choice");
            }

        } while (choice != 5);

        sc.close();
    }
}
```

Output :

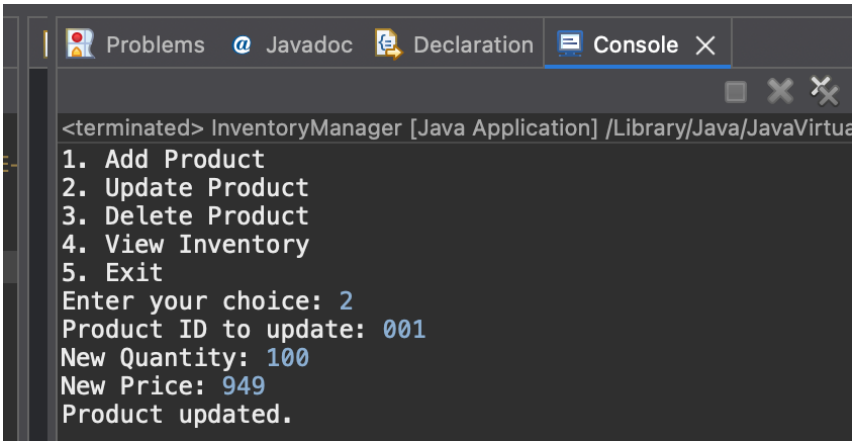
Adding a Product :



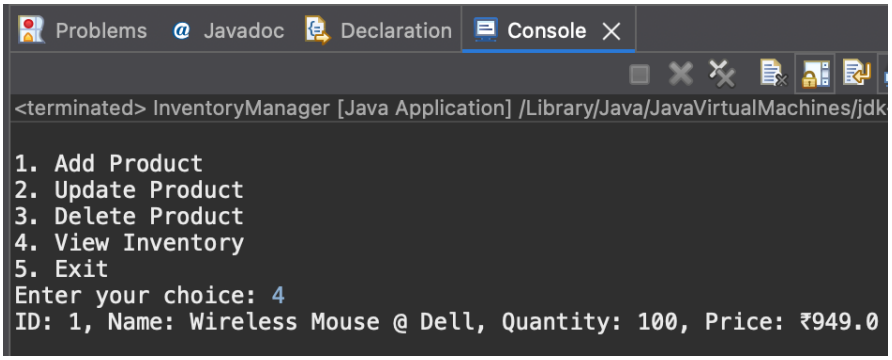
Viewing Inventory :



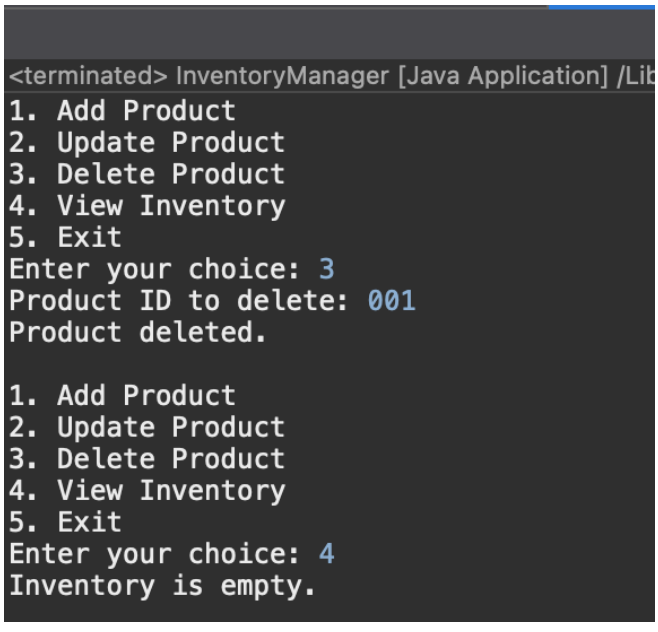
Updating Product data :



Updated Inventory details :



Deleting a product & Viewing the inventory :



Time Complexity Analysis:

- Add Product , Update Product , Delete Product - takes O(1) time complexity, whereas viewing all of them will take O(n) .
- HashMap is already optimal for ID - based access , To sort/view based on name/price,convert inventory.values() to a list and sort using Collections.sort() with a custom comparator.