Exercise 1: Inventory Management System

1. Understand the Problem:

Why Data Structures and Algorithms are Essential in Handling Large Inventories:

- **Efficiency:** Efficient data structures and algorithms are essential for handling large inventories because they ensure quick and reliable operations. Whether you're adding new products, updating existing ones, or deleting old items, these operations need to be performed swiftly to maintain smooth warehouse operations.
- **Scalability:** As the inventory grows, the system must handle the increased load without significant slowdowns. The right data structures and algorithms help achieve this scalability.
- **Organization:** Good data structures help organize the inventory data logically, making it easier to manage and search for specific items.
- **Optimization:** Properly chosen algorithms can optimize resource usage, leading to better performance and cost-effectiveness.

Types of Data Structures Suitable for This Problem:

- **ArrayList:** Useful if the inventory size is relatively static and the most common operations are accessing items by their index.
- **HashMap:** Ideal for fast lookups, insertions, and deletions based on a unique key like productId. It generally offers O(1) time complexity for these operations.
- **LinkedList:** Suitable for scenarios where frequent insertions and deletions are needed, though search times are slower compared to ArrayLists.
- **TreeMap:** Provides ordered data and allows log(n) time complexity for insertion, deletion, and search operations.

2. Setup:

To start, let's set up a new project for the inventory management system. Here’s how you can do it in Java:

1. **Create a New Project:**

Use an IDE like Eclipse or Netbeans. Create a new Java project named InventoryManagementSystem.

2. **Create the Product Class:**

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

    public Product(String productId, String productName, int quantity, double price) {
        this.productId = productId;
        this.productName = productName;
        this.quantity = quantity;
        this.price = price;
    }
    public String getProductId() {
        return productId;
    }
    public void setProductId(String productId) {
        this.productId = productId;
    }
    public String getProductName() {
        return productName;
    }
    public void setProductName(String productName) {
        this.productName = productName;
    }
    public int getQuantity() {
        return quantity;
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
}
```

3. Implementation:

Choosing a Data Structure:

We will use a HashMap to store the products. This allows for fast lookups, additions, and deletions using the productId as the key.

Implementing Methods to Add, Update, and Delete Products:

```
import java.util.HashMap;
public class InventoryManager {
    private HashMap<String, Product> inventory;
    public InventoryManager() {
        inventory = new HashMap<>();
    }
    public void addProduct(Product product) {
        inventory.put(product.getProductId(), product);
    }
    public void updateProduct(String productId, String productName, int quantity, double price) {
        Product product = inventory.get(productId);
        if (product != null) {
            product.setProductName(productName);
            product.setQuantity(quantity);
            product.setPrice(price);
        }
    }
    public void deleteProduct(String productId) {
        inventory.remove(productId);
    }
    public Product getProduct(String productId) {
        return inventory.get(productId);
    }
}
```

4. Analysis:

Time Complexity Analysis:

- **Add Product:**
  - **Time Complexity:** O(1) on average due to the HashMap's ability to perform insertions in constant time.
- **Update Product:**
  - **Time Complexity:** O(1) on average because the HashMap provides constant time complexity for accessing elements by a key.
- **Delete Product:**
  - **Time Complexity:** O(1) on average as the HashMap allows for constant time complexity when removing elements by a key.

Optimizing Operations:

- **Minimizing Collisions in HashMap:**
  - Use a good hash function to minimize collisions, ensuring that the average-case time complexity remains O(1).
- **Rehashing:**
  - Ensure the HashMap is resized and rehashed appropriately to maintain performance as the number of elements grows.
- **Load Factor:**
  - Maintain an optimal load factor (typically 0.75) to balance time complexity and memory usage.

Exercise 2: E-commerce Platform Search Function

1. Understand Asymptotic Notation:

Big O Notation:

- **Definition:** Big O notation describes the upper bound of the time complexity of an algorithm in terms of the input size. It helps in understanding how an algorithm performs as the input size grows.
- **Purpose:** It allows for comparing the efficiency of different algorithms regardless of the hardware or other implementation-specific factors.

Best, Average, and Worst-Case Scenarios:

- **Best Case:** The scenario where the algorithm performs the fewest possible steps. For search operations, this is typically when the desired element is the first one in the data set.
- **Average Case:** The scenario that represents the expected behavior of the algorithm over a wide range of inputs. It often considers the average position of the desired element in the data set.
- **Worst Case:** The scenario where the algorithm performs the maximum number of steps. For search operations, this is when the desired element is the last one or not present in the data set at all.

2. Setup:

Create a Product class with attributes for searching:

```
public class Product {
   private String productId;
   private String productName;
   private String category;
   public Product(String productId, String productName, String category) {
      this.productId = productId;
      this.productName = productName;
      this.category = category;
   }
   public String getProductId() {
      return productId;
   }
   public String getProductName() {
      return productName;
   }
```

```
    public String getCategory() {
        return category;
    }
}
```

3. Implementation:

Linear Search:

```
public class SearchAlgorithms {
    public static Product linearSearch(Product[] products, String targetProductId) {
        for (Product product : products) {
            if (product.getProductId().equals(targetProductId)) {
                return product;
            }
        }
        return null;
    }
}
```

Binary Search:

```
import java.util.Arrays;
import java.util.Comparator;
public class SearchAlgorithms {
    public static Product binarySearch(Product[] products, String targetProductId) {
        int left = 0;
        int right = products.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            int comparison = products[mid].getProductId().compareTo(targetProductId);
            if (comparison == 0) {
                return products[mid];
            }
            if (comparison < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }
}
```

To store products in a sorted array for binary search:

```
Arrays.sort(products, Comparator.comparing(Product::getProductId));
```

4. Analysis:

Time Complexity Comparison:

- **Linear Search:**
  - **Best Case:** O(1) (if the element is the first one)
  - **Average Case:** O(n) (where n is the number of elements)
  - **Worst Case:** O(n) (if the element is the last one or not present)
- **Binary Search:**
  - **Best Case:** O(1) (if the element is at the middle index)
  - **Average Case:** O(log n)
  - **Worst Case:** O(log n)

Which Algorithm is More Suitable:

- **Linear Search:** Suitable for small datasets or unsorted data, as it doesn't require sorting and works directly on the data.
- **Binary Search:** Suitable for larger datasets where the data is sorted. It is significantly faster than linear search for large datasets due to its logarithmic time complexity.

Exercise 3: Sorting Customer Orders

1. Understand Sorting Algorithms:

Bubble Sort:

- **Description:** A simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

- **Time Complexity:** O(n^2) for the average and worst case, O(n) for the best case (when the array is already sorted).
- **Use Case:** Simple to implement but inefficient for large datasets.

Insertion Sort:

- **Description:** Builds the final sorted array one item at a time. It takes each element from the input and inserts it into the correct position within the sorted part of the array.
- **Time Complexity:** O(n^2) for the average and worst case, O(n) for the best case (when the array is already sorted).
- **Use Case:** Efficient for small datasets or nearly sorted arrays.

Quick Sort:

- **Description:** A divide-and-conquer algorithm. It selects a 'pivot' element and partitions the array into two halves, with elements less than the pivot on one side and elements greater than the pivot on the other. It then recursively sorts the two halves.
- **Time Complexity:** O(n log n) on average, O(n^2) in the worst case (when the smallest or largest element is always chosen as the pivot).
- **Use Case:** Generally very efficient and widely used for large datasets.

Merge Sort:

- **Description:** A divide-and-conquer algorithm that divides the array into halves, recursively sorts each half, and then merges the sorted halves to produce the final sorted array.
- **Time Complexity:** O(n log n) for all cases (best, average, worst).
- **Use Case:** Efficient and stable, good for large datasets and linked lists.

2. Setup:

Create a Order class with attributes orderId, customerName, and totalPrice:

```
public class Order {

  private String orderId;

  private String customerName;

  private double totalPrice;

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

    this.orderId = orderId;

    this.customerName = customerName;

    this.totalPrice = totalPrice;

  }

  public String getOrderId() {

    return orderId;
  }
  public String getCustomerName() {
    return customerName;
  }
  public double getTotalPrice() {
    return totalPrice;
  }
}
```

3. Implementation:

Bubble Sort:

```
public class SortingAlgorithms {
    public static void bubbleSort(Order[] orders) {
        int n = orders.length;
        boolean swapped;
        for (int i = 0; i < n - 1; i++) {
            swapped = false;
            for (int j = 0; j < n - 1 - i; j++) {
                if (orders[j].getTotalPrice() > orders[j + 1].getTotalPrice()) {
                    Order temp = orders[j];
                    orders[j] = orders[j + 1];
                    orders[j + 1] = temp;
                    swapped = true;
                }
            }
            if (!swapped) break;
        }
    }
}
```

Quick Sort:

```
public class SortingAlgorithms {
    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].getTotalPrice();
        int i = (low - 1);
        for (int j = low; j < high; j++) {
            if (orders[j].getTotalPrice() <= 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;
    }
}
```

4. Analysis:

Performance Comparison:

- **Bubble Sort:**
    - **Best Case:** O(n) (when the array is already sorted)
    - **Average Case:** O(n^2)
    - **Worst Case:** O(n^2)
- **Quick Sort:**
    - **Best Case:** O(n log n)
    - **Average Case:** O(n log n)

    - **Worst Case:** O(n^2) (when the smallest or largest element is always chosen as the pivot)

Why Quick Sort is Generally Preferred Over Bubble Sort:

- **Efficiency:** Quick Sort is much more efficient than Bubble Sort for large datasets due to its average-case time complexity of O(n log n). Bubble Sort's O(n^2) complexity makes it impractical for large arrays.
- **Performance:** Quick Sort performs well in practice and can be optimized with techniques like choosing a good pivot (e.g., using the median-of-three method).
- **Versatility:** Quick Sort is widely used due to its ability to sort in place and its average-case efficiency. It is typically faster in practice compared to other O(n log n) algorithms like Merge Sort, especially for large datasets.

Exercise 4: Employee Management System

1. Understand Array Representation:

How Arrays are Represented in Memory:

- **Contiguous Memory Allocation:** Arrays are stored in contiguous memory locations. This means that the elements of the array are placed next to each other in memory, which allows for efficient access using index-based operations.
- **Indexing:** Elements in an array can be accessed using their index, with the first element having an index of 0. The address of any element can be calculated using the formula: address = base_address + (index * size_of_element).
- **Advantages:**
    - **Fast Access:** Direct access to elements using their index allows for O(1) time complexity for read and write operations.
    - **Memory Efficiency:** Arrays have a fixed size, which can lead to efficient memory usage if the size is known beforehand.

2. Setup:

Create a class Employee with attributes employeeId, name, position, and salary:

```
public class Employee {

  private String employeeId;

  private String name;
  private String position;
  private double salary;
  public Employee(String employeeId, String name, String position, double salary) {
      this.employeeId = employeeId;
      this.name = name;
      this.position = position;
      this.salary = salary;
  }
  public String getEmployeeId() {
      return employeeId;
  }
  public void setEmployeeId(String employeeId) {
      this.employeeId = employeeId;
  }
  public String getName() {
      return name;
  }
  public void setName(String name) {
      this.name = name;
  }
  public String getPosition() {
      return position;
  }
  public void setPosition(String position) {
      this.position = position;
  }
  public double getSalary() {
      return salary;
  }
  public void setSalary(double salary) {
      this.salary = salary;
  }
```

```
}
```

3. Implementation:

Use an Array to Store Employee Records:

```
public class EmployeeManagementSystem {
    private Employee[] employees;
    private int count;

    public EmployeeManagementSystem(int capacity) {
        employees = new Employee[capacity];
        count = 0;
    }
    public void addEmployee(Employee employee) {
        if (count < employees.length) {
            employees[count] = employee;
            count++;
        } else {
            System.out.println("Array is full, cannot add more employees.");
        }
    }
    public Employee searchEmployee(String employeeId) {
        for (int i = 0; i < count; i++) {
            if (employees[i].getEmployeeId().equals(employeeId)) {
                return employees[i];
            }
        }
        return null;
    }
    public void traverseEmployees() {
        for (int i = 0; i < count; i++) {
            System.out.println(employees[i].getName() + " (" + employees[i].getEmployeeId() + ")");
        }
    }
    public void deleteEmployee(String employeeId) {
        for (int i = 0; i < count; i++) {
            if (employees[i].getEmployeeId().equals(employeeId)) {
                employees[i] = employees[count - 1];  // Replace the found employee with the last
employee
                employees[count - 1] = null;  // Remove the last employee
                count--;
                return;
            }
        }
        System.out.println("Employee not found.");
    }
}
```

4. Analysis:

Time Complexity of Each Operation:

- **Add Employee:**
  - **Time Complexity:** O(1) if there is space in the array.
- **Search Employee:**
  - **Time Complexity:** O(n) in the worst case, where n is the number of employees.
- **Traverse Employees:**
  - **Time Complexity:** O(n) as it needs to visit each element.
- **Delete Employee:**
  - **Time Complexity:** O(n) in the worst case, as it may need to search through all elements to find the employee to delete.

Limitations of Arrays:

- **Fixed Size:** Arrays have a fixed size, which can lead to wasted memory if the array is too large or to overflow if the array is too small.
- **Inefficient Insertions/Deletions:** Inserting or deleting elements can be inefficient as it may require shifting elements to maintain the order.
- **No Dynamic Resizing:** Unlike dynamic data structures like ArrayLists or LinkedLists, arrays cannot grow or shrink in size.

When to Use Arrays:

- **Static Data:** When the size of the dataset is known in advance and is relatively static.

- **Fast Access:** When you need fast access to elements using their index.
- **Memory Constraints:** When memory usage is a concern, as arrays can be more memory efficient if sized correctly.

Exercise 5: Task Management System

1. Understand Linked Lists:

Singly Linked List:

- **Structure:** Consists of nodes where each node contains a data part and a next part that points to the next node in the sequence.
- **Advantages:** Simple structure, efficient for inserting and deleting nodes as it doesn't require shifting elements.
- **Disadvantages:** Only allows traversal in one direction (forward), no direct access to elements by index.

Doubly Linked List:

- **Structure:** Each node contains a data part, a next pointer to the next node, and a prev pointer to the previous node.
- **Advantages:** Allows traversal in both directions (forward and backward), easier to delete a node given only a reference to that node.
- **Disadvantages:** More complex structure, requires more memory due to the extra prev pointers.

2. Setup:

Create a Task class with attributes taskId, taskName, and status:

```
public class Task {
    private String taskId;
    private String taskName;
    private String status;

    public Task(String taskId, String taskName, String status) {
        this.taskId = taskId;
        this.taskName = taskName;
        this.status = status;
    }
    public String getTaskId() {
        return taskId;
    }
    public void setTaskId(String taskId) {
        this.taskId = taskId;
    }
    public String getTaskName() {
        return taskName;
    }
    public void setTaskName(String taskName) {
        this.taskName = taskName;
    }
    public String getStatus() {
        return status;
    }
    public void setStatus(String status) {
        this.status = status;
    }
}
```

3. Implementation:

Singly Linked List:

```
public class TaskManagementSystem {
    private Node head;
    private class Node {
        Task task;
        Node next;
        Node(Task task) {
            this.task = task;
            this.next = null;
        }
    }
```

```
public void addTask(Task task) {
    Node newNode = new Node(task);
    if (head == null) {
        head = newNode;
    } else {
        Node current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }
}
public Task searchTask(String taskId) {
    Node current = head;
    while (current != null) {
        if (current.task.getTaskId().equals(taskId)) {
            return current.task;
        }
        current = current.next;
    }
    return null;
}
public void traverseTasks() {
    Node current = head;
    while (current != null) {
        System.out.println(current.task.getTaskName() + " (" + current.task.getTaskId() + ")");
        current = current.next;
    }
}
public void deleteTask(String taskId) {
    if (head == null) {
        return;
        }
        if (head.task.getTaskId().equals(taskId)) {
            head = head.next;
            return;
        }
        Node current = head;
        while (current.next != null && !current.next.task.getTaskId().equals(taskId)) {
            current = current.next;
        }
        if (current.next != null) {
            current.next = current.next.next;
        }
    }
}
```

4. Analysis:

Time Complexity of Each Operation:

- **Add Task:**
    - **Time Complexity:** O(n) in the worst case (when adding at the end of the list).
- **Search Task:**
    - **Time Complexity:** O(n) (must traverse the list to find the task).
- **Traverse Tasks:**
    - **Time Complexity:** O(n) (must visit each node).
- **Delete Task:**
    - **Time Complexity:** O(n) (must find the task to delete).

Advantages of Linked Lists over Arrays for Dynamic Data:

- **Dynamic Size:** Linked lists can grow and shrink dynamically without the need to allocate a fixed size upfront, unlike arrays which require a predefined size.
- **Efficient Insertions/Deletions:** Adding or removing elements in a linked list can be done efficiently without the need for shifting elements, as is required in arrays.
- **Memory Usage:** Linked lists can be more memory efficient for datasets where the size frequently changes, as they do not require reallocation like dynamic arrays (e.g., ArrayList).

Exercise 6: Library Management System

1. Understand Search Algorithms:

Linear Search:

- **Description:** Linear search is a straightforward algorithm that checks each element in the list one by one until the desired element is found or the list ends.
- **Time Complexity:** O(n) for both the average and worst-case scenarios, where n is the number of elements in the list.
- **Use Case:** Suitable for small or unsorted datasets where the overhead of sorting isn't justified.

Binary Search:

- **Description:** Binary search is an efficient algorithm for finding an item from a sorted list of elements. It repeatedly divides the search interval in half. If the value of the search key is less than the item in the middle of the interval, the algorithm narrows the interval to the lower half. Otherwise, it narrows it to the upper half.
- **Time Complexity:** O(log n) for the average and worst-case scenarios, where n is the number of elements in the list.
- **Use Case:** Suitable for large, sorted datasets due to its logarithmic time complexity.

2. Setup:

Create a Book class with attributes bookId, title, and author:

```
public class Book {
    private String bookId;
    private String title;
    private String author;
    public Book(String bookId, String title, String author) {
        this.bookId = bookId;
        this.title = title;
        this.author = author;
    }
    public String getBookId() {
        return bookId;
    }
    public void setBookId(String bookId) {
        this.bookId = bookId;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
}
```

3. Implementation:

Linear Search to Find Books by Title:

```
public class LibraryManagementSystem {
    private Book[] books;
    private int count;
    public LibraryManagementSystem(int capacity) {
        books = new Book[capacity];
        count = 0;
    }
    public void addBook(Book book) {
        if (count < books.length) {
            books[count] = book;
            count++;
        } else {
            System.out.println("Library is full, cannot add more books.");
        }
    }
    public Book linearSearchByTitle(String title) {
        for (int i = 0; i < count; i++) {
            if (books[i].getTitle().equalsIgnoreCase(title)) {
                return books[i];
            }
        }
        return null;
    }
    public void sortBooksByTitle() {
        Arrays.sort(books, 0, count, Comparator.comparing(Book::getTitle,
String.CASE_INSENSITIVE_ORDER));
    }
    public Book binarySearchByTitle(String title) {
        int left = 0;
        int right = count - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            int comparison = books[mid].getTitle().compareToIgnoreCase(title);
            if (comparison == 0) {
                return books[mid];
            }
            if (comparison < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }
}
```

4. Analysis:

Time Complexity Comparison:

- **Linear Search:**
    - **Best Case:** O(1) (if the first element is the target)
    - **Average Case:** O(n)
    - **Worst Case:** O(n)
- **Binary Search:**
    - **Best Case:** O(1) (if the middle element is the target)
    - **Average Case:** O(log n)
    - **Worst Case:** O(log n)

When to Use Each Algorithm:

- **Linear Search:**
    - **Use Case:** Suitable for small or unsorted datasets.
    - **Advantages:** Simple implementation, no need for pre-sorting.
    - **Disadvantages:** Inefficient for large datasets due to O(n) time complexity.
- **Binary Search:**
    - **Use Case:** Suitable for large, sorted datasets.
    - **Advantages:** Efficient O(log n) time complexity, especially beneficial for large datasets.
    - **Disadvantages:** Requires the dataset to be sorted, additional overhead of sorting if the dataset is not already sorted.

Exercise 7: Financial Forecasting

1. Understand Recursive Algorithms:

Concept of Recursion:

- **Definition:** Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem until it reaches a base case, which terminates the recursive calls.
- **Simplification:** Recursion can simplify the code for problems that have a natural hierarchical structure, such as tree traversal, factorial calculation, and Fibonacci sequence.
- **Base Case and Recursive Case:**
    - **Base Case:** The condition under which the recursion stops.
    - **Recursive Case:** The part of the function where the function calls itself with a smaller or simpler input.

Example: Factorial Calculation:

```
public int factorial(int n) {
   if (n == 0) { // Base case
      return 1;
   }
   return n * factorial(n - 1); // Recursive case
}
```

2. Setup:

Create a method to calculate the future value using a recursive approach.

3. Implementation:

Recursive Algorithm to Predict Future Values:

Let's assume the future value prediction is based on a constant growth rate. The future value FVFVFV at time nnn can be calculated using the formula:

FV(n)=PV×(1+r)nFV(n) = PV \times (1 + r)^nFV(n)=PV×(1+r)n

where:

- PVPVPV is the present value.
- rrr is the growth rate.
- nnn is the number of periods into the future.

Java Implementation:

```
public class FinancialForecasting {
   public double calculateFutureValue(double presentValue, double growthRate, int periods) {
      if (periods == 0) {
         return presentValue;
      }
      return (1 + growthRate) * calculateFutureValue(presentValue, growthRate, periods - 1);
   }

   public static void main(String[] args) {
      FinancialForecasting forecast = new FinancialForecasting();
      double presentValue = 1000.0;
      double growthRate = 0.05;
```

```
        int periods = 10;
        double futureValue = forecast.calculateFutureValue(presentValue, growthRate, periods);
        System.out.println("Future Value: " + futureValue);
    }
}
```

4. Analysis:

Time Complexity of the Recursive Algorithm:

- **Time Complexity:** O(n), where nnn is the number of periods. This is because the function makes one recursive call for each period, resulting in a linear sequence of calls.
- **Space Complexity:** O(n) due to the call stack used by the recursive calls.

Optimizing the Recursive Solution:

- **Memoization:** Store the results of previous computations in a data structure (like an array or a map) to avoid redundant calculations. This reduces the time complexity to O(1) for each memoized result after the initial computation.

Optimized Implementation with Memoization:

```
import java.util.HashMap;
import java.util.Map;
public class FinancialForecastingOptimized {
    private Map<Integer, Double> memo;
    public FinancialForecastingOptimized() {
        memo = new HashMap<>();
    }
    public double calculateFutureValue(double presentValue, double growthRate, int periods) {
        if (periods == 0) {
            return presentValue;
        }
        if (memo.containsKey(periods)) {
            return memo.get(periods);
        }
        double result = (1 + growthRate) * calculateFutureValue(presentValue, growthRate, periods - 1);
        memo.put(periods, result);
        return result;
    }
    public static void main(String[] args) {
        FinancialForecastingOptimized forecast = new FinancialForecastingOptimized();
        double presentValue = 1000.0;
        double growthRate = 0.05;
        int periods = 10;
        double futureValue = forecast.calculateFutureValue(presentValue, growthRate, periods);
        System.out.println("Future Value: " + futureValue);
    }
}
```