Answers to Data Structures and Algorithms Exercises

# Exercise 1: Inventory Management System

1. Data structures and algorithms are essential for efficiently storing, retrieving, and updating product data. HashMap is suitable for fast lookups and updates using productId.  
2. Class Product: Contains productId, productName, quantity, and price.  
3. Use HashMap<String, Product> for storage. Implement methods: addProduct(), updateProduct(), deleteProduct().  
4. Time Complexity:  
 - Add: O(1)  
 - Update: O(1)  
 - Delete: O(1)  
 Optimization: Ensure unique keys and proper hash function.

# Exercise 2: E-commerce Platform Search Function

1. Big O notation expresses algorithm efficiency. It includes:  
 - Best: Ideal case  
 - Average: Normal case  
 - Worst: Maximum effort needed  
2. Class Product: Includes productId, productName, and category.  
3. Linear Search: O(n), iterates through each item.  
 Binary Search: O(log n), on sorted data only.  
4. Binary Search is more efficient but requires sorted data. Use for large, sorted datasets.

# Exercise 3: Sorting Customer Orders

1. Bubble Sort: O(n^2), simple but slow.  
 Quick Sort: O(n log n) average, fast for large datasets.  
2. Class Order: orderId, customerName, totalPrice.  
3. Bubble Sort: Compare and swap adjacent items.  
 Quick Sort: Divide and conquer using pivot.  
4. Quick Sort is preferred due to better average performance.

# Exercise 4: Employee Management System

1. Arrays store data in contiguous memory, allowing index-based access.  
2. Class Employee: employeeId, name, position, salary.  
3. Use Employee[] array. Methods: addEmployee(), searchEmployee(), traverseEmployees(), deleteEmployee().  
4. Time Complexity:  
 - Add: O(1) if space available  
 - Search: O(n)  
 - Traverse: O(n)  
 - Delete: O(n) (shift elements)  
 Limitation: Fixed size, inefficient inserts/deletes.

# Exercise 5: Task Management System

1. Linked Lists: Dynamic memory, ideal for frequent inserts/deletes.  
 - Singly: One direction  
 - Doubly: Two-way traversal  
2. Class Task: taskId, taskName, status.  
3. Implement SinglyLinkedList with Node<Task>. Methods: addTask(), searchTask(), traverseTasks(), deleteTask().  
4. Time Complexity:  
 - Add: O(1) at head, O(n) at end  
 - Search: O(n)  
 - Traverse: O(n)  
 - Delete: O(n)  
 Advantage: Efficient dynamic memory handling.

# Exercise 6: Library Management System

1. Linear Search: O(n), unsorted data.  
 Binary Search: O(log n), sorted data.  
2. Class Book: bookId, title, author.  
3. Use Book[] array.  
 - Linear: Loop through each element.  
 - Binary: Divide search space iteratively or recursively.  
4. Use Linear for small/unsorted data.  
 Use Binary for large/sorted data for faster lookup.

# Exercise 7: Financial Forecasting

1. Recursion: Function calls itself for subproblems. Simple and elegant for divide-and-conquer.  
2. Recursive function: futureValue(year) = baseValue \* (1 + growthRate)^year.  
3. Implement recursive method for futureValue(year).  
4. Time Complexity: O(n) for each year. Optimize using memoization or convert to iteration for large data.