Solutions for Algorithms and Data Structures Exercises

# Exercise 1: Inventory Management System

Importance: Data structures and algorithms are essential for fast and efficient operations like adding, updating, or deleting items in large inventories.

Suitable Data Structures: HashMap (for fast retrieval by key), ArrayList (for simple list management).

Class Design:  
class Product {  
 public int ProductId;  
 public string ProductName;  
 public int Quantity;  
 public double Price;  
}  
```

Storage: List<Product> or Dictionary<int, Product>.

Methods:

- Add: inventory.Add(productId, new Product(...));

- Update: inventory[productId].Quantity += 5;

- Delete: inventory.Remove(productId);

Time Complexities:

- Add: O(1) for Dictionary, O(n) for List (if search needed).

- Update/Delete: O(1) for Dictionary, O(n) for List.

Optimization: Use Dictionary for frequent access by productId.

# Exercise 2: E-commerce Platform Search Function

Big O Notation: Describes the performance (complexity) of an algorithm.

Best/Average/Worst Case:

- Linear Search: Best O(1), Worst O(n)

- Binary Search: Best O(1), Worst O(log n)

Class:  
class Product {  
 public int ProductId;  
 public string ProductName;  
 public string Category;  
}  
```

Linear Search: Iterate over array.

Binary Search: Requires sorted array; use divide and conquer.

Comparison: Binary search is faster for large sorted arrays.

# Exercise 3: Sorting Customer Orders

Sorting Algorithms:

- Bubble Sort: Repeatedly swaps adjacent elements.

- Quick Sort: Divide and conquer; partitions the array.

- Merge Sort: Recursive divide and merge.

- Insertion Sort: Inserts elements into correct position.

Class:  
class Order {  
 public int OrderId;  
 public string CustomerName;  
 public double TotalPrice;  
}  
```

Bubble Sort Implementation (C#):  
void BubbleSort(List<Order> orders) {  
 for (int i = 0; i < orders.Count - 1; i++)  
 for (int j = 0; j < orders.Count - i - 1; j++)  
 if (orders[j].TotalPrice > orders[j + 1].TotalPrice) {  
 var temp = orders[j];  
 orders[j] = orders[j + 1];  
 orders[j + 1] = temp;  
 }  
}

Quick Sort: O(n log n), Bubble Sort: O(n^2).

Quick Sort is preferred for efficiency.

# Exercise 4: Employee Management System

Array Representation: Contiguous memory, fast access using index.

Class:

class Employee {  
 public int EmployeeId;  
 public string Name;  
 public string Position;  
 public double Salary;  
}  
```

Operations:

- Add: Insert at available index.

- Search: Linear scan.

- Traverse: Loop through array.

- Delete: Set index to null or shift elements.

Time Complexities: Add/Search/Delete: O(n), Traverse: O(n).

Limitations: Fixed size, costly insertions/deletions.

# Exercise 5: Task Management System

Linked Lists: Dynamic memory, better insert/delete.

- Singly: One direction.

- Doubly: Forward/backward traversal.

Class:

class Task {  
 public int TaskId;  
 public string TaskName;  
 public string Status;  
 public Task Next;  
}

Operations: Add/Search/Traverse/Delete implemented via pointer updates.

Time Complexity: O(n) for most operations.

Advantages: Dynamic size, efficient insert/delete.

# Exercise 6: Library Management System

Search Algorithms:

- Linear: Check every element.

- Binary: Divide and conquer (needs sorted array).

Class:

class Book {  
 public int BookId;  
 public string Title;  
 public string Author;  
}

Linear Search: O(n), Binary Search: O(log n).

Use Binary Search for large, sorted datasets.

# Exercise 7: Financial Forecasting

Recursion: Function calls itself to solve subproblems.

Example Recursive Function:  
double PredictGrowth(double current, double rate, int years) {  
 if (years == 0) return current;  
 return PredictGrowth(current \* (1 + rate), rate, years - 1);  
}

Time Complexity: O(n).

Optimization: Use memoization or loop to avoid repeated calls.