**Week\_1\_Assignments**

**Exercise 1: Inventory Management System**

Why Data Structures and Algorithms are Essential:

* **Efficiency:** Efficient data structures and algorithms ensure that operations such as adding, updating, retrieving, and deleting products are performed quickly, which is crucial for large inventories.
* **Scalability:** As the size of the inventory grows, efficient data structures can handle the increased load without significant performance degradation.
* **Memory Management:** Proper data structures help manage memory efficiently, preventing excessive use of resources.
* **Ease of Maintenance:** Well-chosen data structures and algorithms make the codebase easier to understand, maintain, and extend.

Types of Data Structures Suitable for Inventory Management

* **ArrayList:** Provides fast access and update operations but slow insertions and deletions if not at the end of the list.
* **HashMap:** Allows for fast access, insertions, and deletions by using a key-value pair, where the key can be the product ID.
* **LinkedList:** Efficient for insertions and deletions at any point in the list but slower access time compared to an array.
* **TreeMap:** Keeps the keys in a sorted order and provides log(n) time complexity for basic operations.

Time Complexity Analysis:

* **Add Product:**

Time Complexity: O(1)

Explanation: Adding a product to a HashMap is an average O(1) operation due to the underlying hash table implementation.

* **Update Product:**

Time Complexity: O(1)

Explanation: Updating a product in a HashMap is also an O(1) operation as it involves replacing the value for a given key.

* **Delete Product:**

Time Complexity: O(1)

Explanation: Deleting a product from a HashMap is an O(1) operation since it involves removing the key-value pair.

* **Retrieve Product:**

Time Complexity: O(1)

Explanation: Retrieving a product by its ID is an O(1) operation due to direct access using the key.

Optimization Strategies:

* **Rehashing:** Ensure that the HashMap has an appropriate load factor to minimize collisions and maintain O(1) performance.
* **Concurrency:** If the inventory system needs to support concurrent access, consider using a ConcurrentHashMap.
* **Validation:** Implement validation checks to ensure product data integrity before adding or updating products in the inventory.

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

**Big O Notation:**

Big O notation is a mathematical notation used to describe the upper bound of an algorithm's running time or space requirements in terms of the size of the input. It characterises the performance of an algorithm by expressing its complexity as a function of the input size, ignoring constants and lower-order terms. This helps in comparing the efficiency of different algorithms, especially for large inputs.

* **O(1):** Constant time – The running time is independent of the input size.
* **O(log n):** Logarithmic time – The running time grows logarithmically with the input size.
* **O(n):** Linear time – The running time grows linearly with the input size.
* **O(n log n):** Linearithmic time – The running time grows as a product of n and log n.
* **O(n^2):** Quadratic time – The running time grows quadratically with the input size.

**Best Case:** The scenario where the algorithm performs the minimum number of operations. For search operations, this often occurs when the element being searched for is the first element in the data structure.

**Average Case:** The expected scenario, considering all possible inputs of the same size. It represents the average performance of the algorithm over all inputs.

**Worst Case:** The scenario where the algorithm performs the maximum number of operations. For search operations, this often occurs when the element being searched for is not present in the data structure or is the last element.

#### **Linear Search:**

* **Best Case:** O(1) – The element is the first in the list.
* **Average Case:** O(n) – On average, the element is found in the middle of the list.
* **Worst Case:** O(n) – The element is the last in the list or not present.

#### **Binary Search (on sorted data):**

* **Best Case:** O(1) – The element is the middle element.
* **Average Case:** O(log n) – The search space is halved with each step.
* **Worst Case:** O(log n) – The element is not present or at one of the ends of the list.

**Binary Search is Suitable**

For an e-commerce platform, where the number of products can be very large, **binary search** is generally more suitable due to its logarithmic time complexity, which ensures faster search performance compared to linear search. However, this requires maintaining a sorted list of products, which can be managed with efficient sorting algorithms or using data structures that keep elements sorted, such as balanced trees or heaps. If real-time updates and insertions are frequent, a more advanced data structure like a hash table or an indexed database may be used for even better performance.

**Exercise 3: Sorting Customer Orders**

#### **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. The pass through the list is repeated until the list is sorted.
* **Time Complexity:**
  + Best Case: O(n)
  + Average Case: O(n^2)
  + Worst Case: O(n^2)

**Insertion Sort:**

* **Description:** Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list.
* **Time Complexity:**
* Best Case: O(n)
* Average Case: O(n^2)
* Worst Case: O(n^2)

**Quick sort:**

* **Description:** An efficient, in-place, comparison-based sorting algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
* **Time Complexity:**
* Best Case: O(n log n)
* Average Case: O(n log n)
* Worst Case: O(n^2) (but can be improved with good pivot selection strategies)

**Merge Sort:**

* **Description:** A divide-and-conquer algorithm that divides the unsorted list into n sublists, each containing one element, then repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining.
* **Time Complexity:**
* Best Case: O(n log n)
* Average Case: O(n log n)
* Worst Case: O(n log n)

#### **Quick Sort is Generally Preferred Over Bubble Sort:**

* **Efficiency:** Quick sort is generally much faster than bubble sort, especially for large datasets. Its average and best-case time complexities are O(n log n), while bubble sort has an average and worst-case time complexity of O(n^2).
* **Scalability:** Quick sort scales better with increasing data sizes compared to bubble sort.
* **Real-world Usage:** Quick sort is widely used in practice due to its superior performance, despite its worst-case scenario, which can be mitigated with good pivot selection strategies (e.g., choosing a random pivot, median-of-three rule).

**Exercise 4: Employee Management System**

#### **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 stored one after the other in memory.
* **Indexing:** Arrays provide constant-time access (O(1)) to elements using indices. This is because the memory address of any element can be calculated directly using the base address of the array and the index.
* **Fixed Size:** Arrays have a fixed size, which means the size of the array must be known at compile time (for static arrays) or at the time of allocation (for dynamic arrays).

#### 

#### **Advantages of Arrays:**

* **Fast Access:** Constant-time access to elements using indices.
* **Cache-Friendly:** Contiguous memory allocation improves cache performance.
* **Simple Data Structure:** Easy to use and implement.

#### **Time Complexity of Each Operation:**

* **Add Employee:**
  + Time Complexity: O(1) – Adding an employee to the end of the array is a constant-time operation as long as the array is not full.
* **Search Employee:**
  + Time Complexity: O(n) – In the worst case, the algorithm must search through the entire array to find the employee.
* **Traverse Employees:**
  + Time Complexity: O(n) – Traversing the array involves visiting each element once.
* **Delete Employee:**
  + Time Complexity: O(n) – In the worst case, the algorithm must shift all elements after the deleted employee to fill the gap.

#### **Limitations of Arrays:**

* **Fixed Size:** Once an array is created, its size cannot be changed. This can lead to wasted memory if the array is too large or overflow if it is too small.
* **Inefficient Deletion and Insertion:** Deleting or inserting an element (other than at the end) requires shifting elements, which can be inefficient for large arrays.
* **Static Nature:** Arrays do not provide dynamic memory allocation, which can be a limitation when the number of elements is not known in advance.

#### **When to Use Arrays:**

* **When the Number of Elements is Known:** Arrays are suitable when the number of elements is known and does not change frequently.
* **For Fast Access:** Arrays provide O(1) time complexity for accessing elements by index, making them ideal for situations where fast access is required.
* **Simple Data Structure:** Arrays are easy to use and understand, making them suitable for simple applications with a fixed number of elements.

**Exercise 5: Task Management System**

#### **Types of Linked Lists:**

**Singly Linked List:**

* Structure: Each node contains data and a reference to the next node in the sequence.
* Advantages: Simple to implement and uses less memory per node compared to doubly linked lists.
* Disadvantages: Cannot easily traverse backward and requires more time to search for an element compared to arrays.

**Doubly Linked List:**

* **Structure:** Each node contains data, a reference to the next node, and a reference to the previous node.
* **Advantages:** Can be traversed in both directions, making certain operations like deletions and insertions more efficient.
* **Disadvantages:** Requires more memory per node due to the additional reference and is slightly more complex to implement.

#### **Time Complexity of Each Operation:**

* **Add Task:**
  + Time Complexity: O(n) – Adding a task involves traversing to the end of the list, which takes O(n) time in the worst case.
* **Search Task:**
  + Time Complexity: O(n) – Searching for a task involves traversing the list to find the task, which takes O(n) time in the worst case.
* **Traverse Tasks:**
  + Time Complexity: O(n) – Traversing the list involves visiting each node, which takes O(n) time.
* **Delete Task:**
  + Time Complexity: O(n) – Deleting a task involves searching for the task and then removing it, which takes O(n) time in the worst case.

#### **Advantages of Linked Lists Over Arrays for Dynamic Data:**

* **Dynamic Size:** Linked lists can grow and shrink dynamically, whereas arrays have a fixed size.
* **Efficient Insertions/Deletions:** Insertions and deletions in a linked list are more efficient compared to arrays, especially when dealing with large datasets or when operations are performed in the middle of the list.
* **Memory Utilization:** Linked lists allocate memory as needed, which can be more efficient than arrays, which allocate a fixed amount of memory upfront.

**Exercise 6: Library Management System**

**Linear Search**:

* **Description:** Linear search sequentially checks each element of the list until it finds the target element or reaches the end of the list.
* **Time Complexity:**
* Best Case: O(1) – The target element is the first element in the list.
* Average Case: O(n) – The target element is somewhere in the middle of the list.
* Worst Case: O(n) – The target element is the last element or not present in the list.

**Binary Search:**

* **Description:** Binary search works on sorted lists by repeatedly dividing the search interval in half. If the target value is less than the middle element, the search continues in the lower half; otherwise, it continues in the upper half.
* **Time Complexity:**
* Best Case: O(1) – The target element is the middle element of the list.
* Average Case: O(log n) – The search interval is halved with each step.
* Worst Case: O(log n) – The target element is at one end of the list or not present.

#### **When to Use Each Algorithm:**

* **Linear Search:**
  + Use when the list is unsorted or when the list is small.
  + Simple to implement and does not require the list to be sorted.
  + Suitable for scenarios where elements are frequently added or removed, making sorting impractical.
* **Binary Search:**
  + Use when the list is sorted and the size of the dataset is large.
  + Much faster than linear search for large datasets due to its O(log n) time complexity.
  + Requires the list to be sorted, so additional time may be needed to sort the list before performing the search.

**Exercise 7: Financial Forecasting**

**Concept of Recursion:**

* **Definition:** Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem.
* **Base Case:** This is a condition that stops the recursion. Without a base case, the function would call itself indefinitely, leading to a stack overflow.
* **Recursive Case:** This is the part of the function where the function calls itself with a smaller or simpler input.

**Benefits of Recursion:**

* **Simplifies Code:** Recursion can make the code more concise and easier to read, especially for problems that naturally fit a recursive approach (e.g., factorial, Fibonacci sequence, tree traversal).
* **Divide and Conquer:** Recursion can effectively break down complex problems into simpler subproblems, which are easier to solve.

**Time Complexity of Recursive Algorithm:**

* **Time Complexity:** O(n) – The algorithm makes one recursive call for each period, leading to linear time complexity with respect to the number of periods.
* **Space Complexity:** O(n) – Each recursive call adds a new frame to the call stack, so the space complexity is also linear.

**Optimising the Recursive Solution:**

To avoid excessive computation and potential stack overflow for large inputs, we can use an iterative approach or memoization.

**Iterative Approach:**

* Time Complexity: O(n) – Iterates through each period.
* Space Complexity: O(1) – Uses a constant amount of space.

**Memoization Approach:**

* **Time Complexity:** O(n) – Each period is computed once and stored.
* **Space Complexity:** O(n) – Stores results for each period up to n.