## Exercise 1 :-

Data structures and algorithms are essential for efficiently managing large inventories because they decide how data is stored, accessed, and manipulated. Efficient data structures ensure quick data retrieval, modification, and storage, which are crucial in a warehouse setting where operations need to be fast and accurate to maintain optimal inventory levels and support logistics operations.

The types of data structures suitable for this problem are :-

* **ArrayList**: Good for simple storage with fast access and iteration, but slow for deletions and insertions in the middle.
* **HashMap**: Provides average O(1) time complexity for insertions, deletions, and lookups, making it highly suitable for inventory management where fast access to product information is critical.

Analysis :-

#### **Time Complexity**

* **Add Operation** :- O(1) average case due to the direct access provided by the hash map.
* **Update Operation** :- O(1) average case, since it involves a single lookup and modification.
* **Delete Operation** :- O(1) average case, as it involves a single lookup and removal.

## Exercise 2 :-

#### **Big O Notation**

Big O notation is a mathematical representation used to describe the performance or complexity of an algorithm as the input size grows. It provides an upper bound on the time or space complexity, allowing us to focus on the most significant factors affecting an algorithm's efficiency.

* **O(1)** :- Constant time. The operation takes the same amount of time regardless of the input size.
* **O(log n)** :- Logarithmic time. The time increases logarithmically as the input size increases. Common in algorithms that divide the problem in half each step, such as binary search.
* **O(n)** :- Linear time. The time increases linearly with the input size. Common in algorithms that process each input element, such as linear search.
* **O(n log n)** :- Linearithmic time. The time complexity of efficient sorting algorithms like merge sort and quicksort.
* **O(n^2)** :- Quadratic time. The time increases quadratically with the input size. Common in algorithms with nested loops, like bubble sort.

#### **Best, Average, and Worst-Case Scenarios**

* **Best Case** :- The scenario where the algorithm performs the minimum number of operations.
* **Average Case** :- The scenario representing the average number of operations an algorithm performs across all inputs of the same size.
* **Worst Case** :- The scenario where the algorithm performs the maximum number of operations.

#### **Analysis**

For search operations:

* **Linear Search** :- Best O(1), Average O(n/2), Worst O(n)
* **Binary Search** :- Best O(1), Average O(log n), Worst O(log n)

**Binary Search** is more suitable due to its logarithmic time complexity, which provides much faster search performance in large inventories. The overhead of sorting the product list is justified by the significantly reduced search times for repeated queries.

## Exercise 3 :-

#### **Bubble Sort**

* A simple comparison-based sorting algorithm. It 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** :-
  + Best Case :- O(n)
  + Average Case :- O(n^2)
  + Worst Case :- O(n^2)

#### **Insertion Sort**

* Builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.
* **Time Complexity** :-
  + Best Case :- O(n)
  + Average Case :- O(n^2)
  + Worst Case :- O(n^2)

#### **Quick Sort**

* A highly efficient 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. The sub-arrays are then sorted recursively.
* **Time Complexity** :-
  + Best Case :- O(n log n)
  + Average Case :- O(n log n)
  + Worst Case :- O(n^2)

#### **Merge Sort**

* A divide and conquer algorithm. It divides the unsorted list into n sublists, each containing one element, and 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)

#### **Time Complexity Comparison**

* **Bubble Sort**:
  + **Best Case** :- O(n) - When the array is already sorted.
  + **Average Case** :- O(n^2) - Typical case for an unsorted array.
  + **Worst Case** :- O(n^2) - When the array is sorted in reverse order.
* **Quick Sort**:
  + **Best Case** :- O(n log n) - Ideal scenario where the pivot splits the array evenly.
  + **Average Case** :- O(n log n) - Average case scenario with decent pivots.
  + **Worst Case** :- O(n^2) - Worst case when the smallest or largest element is always picked as pivot.

Quick Sort is preferred over Bubble Sort because :-

* + **Efficiency** :- Quick Sort is much more efficient for larger data sets due to its average O(n log n) time complexity compared to Bubble Sort's O(n^2).
  + **Practical Performance** :- Quick Sort generally performs better in practice due to fewer swaps and comparisons on average.
  + **Scalability** :- Quick Sort scales better with larger data sets, making it suitable for applications like an e-commerce platform where the number of orders can be very large.

## Exercise 4 :-

#### **Arrays in Memory**

* **Memory Layout** :- Arrays are stored in contiguous memory locations, which means each element in the array is located next to its neighbor. This allows for efficient indexing and retrieval of elements.
* **Indexing** :- Accessing any element in the array can be done in constant time O(1), because the address of any element can be calculated directly using its index and the base address of the array.
* **Advantages** :-
  + **Fast Access** :- Due to direct indexing, accessing any element is very fast.
  + **Predictable Memory Usage** :- The size of the array is fixed, making memory usage predictable.
  + **Cache-Friendly** :- Contiguous memory layout improves cache performance.

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

* **Add** :-
  + **Best Case** :- O(1) - When the array is not full.
  + **Worst Case** :- O(1) - The array is full, and the operation fails.
* **Search** :-
  + **Best Case** :- O(1) - When the employee is at the first position.
  + **Worst Case** :- O(n) - When the employee is at the last position or not present.
* **Traverse** :-
  + **Best Case** :- O(n) - All elements are traversed.
  + **Worst Case** :- O(n) - All elements are traversed.
* **Delete** :-
  + **Best Case** :- O(1) - When the employee is at the last position.
  + **Worst Case** :- O(n) - When the employee is at the first position, requiring all subsequent elements to be shifted.

#### **Limitations of Arrays**

* + **Fixed Size** :- Arrays cannot change size dynamically, leading to either wasted space or insufficient space.
  + **Inefficient Deletions** :- Deleting an element requires shifting all subsequent elements, which can be time-consuming.
  + **Insertion Complexity** :- Inserting an element in the middle of the array requires shifting elements to make space.

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

* + **When the Size is Known** :- Arrays are suitable when the number of elements is known and fixed.
  + **When Direct Access is Required** :- Arrays are efficient for scenarios requiring frequent access to elements by index.
  + **When Memory Efficiency is Crucial** :- Arrays have minimal memory overhead compared to other data structures like linked lists.

## Exercise 5 :-

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

1. **Singly Linked List** :-
   * **Structure** :- Each node contains data and a reference (or pointer) to the next node in the sequence.
   * **Operations** :- Easy to traverse forward, but can't traverse backward. Simpler to implement compared to doubly linked lists.
   * **Use Cases** :- When memory usage is a concern, and operations involve primarily forward traversals.
2. **Doubly Linked List**:
   * **Structure** :- Each node contains data, a reference to the next node, and a reference to the previous node.
   * **Operations** :- Can be traversed both forward and backward. Slightly more complex and uses more memory due to the additional reference.
   * **Use Cases** :- When frequent insertions and deletions at both ends are required.

#### **Time Complexity of Operations**

* **Add Task** :-
  + **Time Complexity** :- O(n) - Since we need to traverse to the end of the list to add a new task.
* **Search Task** :-
  + **Time Complexity** :- O(n) - In the worst case, we might need to traverse the entire list.
* **Traverse Tasks** :-
  + **Time Complexity** :- O(n) - We need to visit each node once.
* **Delete Task** :-
  + **Time Complexity** :- O(n) - In the worst case, we might need to traverse the entire list to find and delete a task.

#### **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** :- Linked lists allow efficient insertions and deletions, especially when dealing with nodes at the beginning or middle of the list, without needing to shift elements as in arrays.
* **Memory Utilization** :- Linked lists can utilize memory more efficiently for dynamic datasets, avoiding the need to allocate large contiguous memory blocks upfront as required by arrays

## Exercise 6 :-

#### **Linear Search**

* Linear search iterates through the list element by element, checking each one until it finds the target or reaches the end of the list.
* **Time Complexity** :- O(n), where n is the number of elements in the list.
  + **Best Case** :- O(1) - The target element is the first element.
  + **Worst Case** :- O(n) - The target element is the last element or not present in the list.

#### **Binary Search**

* Binary search works on sorted lists by repeatedly dividing the search interval in half. It compares the target value with the middle element, and if they are not equal, it eliminates half of the search space.
* **Time Complexity** :- O(log n), where n is the number of elements in the list.
  + **Best Case** :- O(1) - The target element is the middle element of the list.
  + **Worst Case** :- O(log n) - The target element is at the start or end of the list.

#### **Time Complexity Comparison**

* **Linear Search** :-
  + **Best Case** :- O(1) - The target element is the first element.
  + **Average Case** :- O(n) - The target element is somewhere in the middle.
  + **Worst Case** :- O(n) - The target element is the last element or not present.
* **Binary Search** :-
  + **Best Case** :- O(1) - The target element is the middle element of the list.
  + **Average Case** :- O(log n) - The target element is somewhere in the list.
  + **Worst Case** :- O(log n) - The target element is at the start or end of the list.

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

* **Linear Search** :-
  + Suitable for small datasets where the overhead of sorting does not justify the performance gains of binary search.
  + Useful when the list is unsorted, and there's no need to sort it for other purposes.
  + Easy to implement and understand.
* **Binary Search** :-
  + Suitable for large datasets where the list is sorted or needs to be sorted for other operations.
  + Provides significant performance improvements for search operations in sorted lists.
  + More complex to implement compared to linear search but offers better performance for large, sorted datasets.

## Exercise 7 :-

## **Recursion**

* Recursion is a method of solving a problem where the solution involves solving smaller instances of the same problem.
* **Base Case** :- This is the condition in which the recursive calls will stop. Without a base case, the recursion would continue indefinitely.
* **Recursive Case** :- This is the part of the function which breaks down the problem into smaller instances and calls itself.

#### **Time Complexity**

* **Time Complexity** :- O(n), where n is the number of years. This is because the function makes one recursive call per year until it reaches the base case.

#### **Optimization to Avoid Excessive Computation**

* **Memorization** :- Store the results of intermediate calculations to avoid redundant computations in subsequent recursive calls.
* **Iterative Approach** :- Convert the recursive solution to an iterative one to avoid the overhead of recursive function calls and reduce the risk of stack overflow for large values of n.