**ANALYSIS**

***1) Time Complexity Analysis: O(1) on average for Add Operation because of HashMap's hashing technique.***

***Update Operation: O(1) on average, if you are aware of the productId and the efficiency of HashMap.***

***Delete Operation: HashMap averages O(1).***

***Enhancements:***

***HashMap Resizing: To prevent repeated resizing, make sure the initial capacity and load factor are set correctly.***

***Collisions: Use strong hash functions and handle collisions inside HashMap to effectively handle hash collisions (e.g., utilizing linked lists or trees for buckets).***

***Caching: To speed up retrieval times, use caching for products that are accessed frequently.***

***You may guarantee fast access and modification times by employing a hash map for the inventory, which makes the system effective and scalable for handling a big inventory.***

***Analysis***

1) Comparison of Analysis Time Complexity: Linear Search: Best Case: O(1)

Mean Situation: O(n)

The worst case is O(n).

Best Case for Binary Search: O(1)

Mean Situation: O(log n)

Odds of Success: O(log n)

Which Algorithm Works Best?

For tiny datasets or unsorted data when sorting overhead is not warranted, linear search is appropriate.

For huge datasets that can be maintained sorted, binary search is perfect. Since its temporal complexity is logarithmic, it is more effective for larger arrays. But in order to use it, the data must be sorted, which could add to the overhead.

For an e-commerce site, a larger dataset with sorted product storage is usually more suited for Binary Search due to its faster search times than linear search. For scenarios or smaller datasets where sorting.

Performance Evaluation:

Bubble Sorting

Time Complexity: In typical and worst situations, O(n²). Because to its quadratic time complexity, it is inefficient for large datasets.

Fast Sort:

Compared to Bubble Sort, Time Complexity is significantly more efficient for large datasets, averaging O(n log n). Although its worst-case complexity is O(n²), this can be lessened by employing effective pivot selection techniques (such as median-of-three or random pivot).

Reasons for Generally Preferring Quick Sort:

Efficiency: Since Bubble Sort has an average-case time complexity of O(n log n), which is far worse than Quick Sort's O(n²), Quick Sort is usually chosen.

Practical Performance: Quick Sort's divide-and-conquer strategy and effective partitioning allow it to perform exceptionally well in practice, even with its worst-case time complexity.

Quick Sort is an in-place sorting method.

1. Analysis  
   Time Complexity Analysis:  
     
   Add Operation:  
     
   Best Case: O(1) (when the list is empty or adding to the end of the list)  
   Average Case: O(n) (when adding to the end of the list and needs to traverse)  
   Worst Case: O(n) (if the list is large and traversal is required)  
   Search Operation:  
     
   Best Case: O(1) (when the task is at the head)  
   Average Case: O(n) (linear search through the list)  
   Worst Case: O(n) (if the task is at the end or not found)  
   Traverse Operation:  
     
   Time Complexity: O(n) (each node needs to be accessed and printed)  
   Delete Operation:  
     
   Best Case: O(1) (when the task is at the head)  
   Average Case: O(n) (linear search to find the task and delete it)  
   Worst Case: O(n) (if the task is at the end or not found)  
   Advantages of Linked Lists Over Arrays for Dynamic Data:  
     
   Dynamic Size: Linked lists can easily grow or shrink in size without the need for resizing or reallocating memory.  
   Efficient Insertions/Deletions: Inserting or deleting tasks does not require shifting elements as in arrays. This makes linked lists more efficient for scenarios where frequent insertions and deletions occur.  
   Limitations:  
     
   Memory Overhead: Each node in a linked list requires extra memory for the pointer.  
   No Direct Access: Linked lists do not support direct access to elements, making certain operations slower compared to arrays.
2. Analysis  
   Time Complexity Comparison:  
     
   Linear Search:  
     
   Best Case: O(1)  
   Average Case: O(n)  
   Worst Case: O(n)  
   Binary Search:  
     
   Best Case: O(1)  
   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 where sorting is not feasible or needed. It works well when the dataset is relatively small and simplicity is preferred.  
   Binary Search:  
   Use Case: Best used with large, sorted datasets due to its efficient O(log n) time complexity. It is ideal when the dataset is static or changes infrequently, allowing it to remain sorted.
3. Time Complexity of Recursive Algorithm:  
     
   Time Complexity: O(n), where n is the number of periods. Each recursive call reduces the problem size by one, so the algorithm makes n recursive calls.  
   Space Complexity: O(n), due to the call stack storing n frames in the worst case. Each recursive call adds a frame to the stack.  
   Optimization to Avoid Excessive Computation:  
     
   Memoization: Store the results of previously computed values to avoid redundant calculations. This technique can be applied to improve efficiency in cases where the problem involves overlapping sub-problems.