1. Analysis  
   Time Complexity Analysis:  
   Add Operation: O(1) on average due to HashMap's hashing mechanism.  
   Update Operation: O(1) on average, assuming you know the productId and HashMap performs efficiently.  
   Delete Operation: O(1) on average for HashMap.  
   Optimizations:  
   HashMap Resizing: Ensure that the initial capacity and load factor are set appropriately to avoid frequent resizing.  
   Collisions: Handle hash collisions efficiently by using good hash functions and handling collisions within HashMap (e.g., using linked lists or trees for buckets).  
   Caching: Implement caching for frequently accessed products to improve retrieval time.  
   By using a HashMap for the inventory, you ensure quick access and modification times, making the system scalable and efficient for managing a large inventory.
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)  
   Which Algorithm is More Suitable?  
   Linear Search: Suitable for small datasets or unsorted data where sorting overhead is not justified.  
   Binary Search: Ideal for large datasets that can be kept sorted. It is more efficient for larger arrays due to its logarithmic time complexity. However, it requires the data to be sorted, which might involve additional overhead for sorting.  
   For an e-commerce platform:  
   Binary Search is typically preferred for large datasets where products are stored in a sorted manner because it provides faster search times compared to linear search. For smaller datasets or scenarios where sorting is not feasible, Linear Search might be more practical.  
   Summary:  
   Linear Search is simple but less efficient for large datasets.  
   Binary Search is more efficient for large, sorted datasets but requires pre-sorting.
3. Analysis  
   Performance Comparison:  
     
   Bubble Sort:  
     
   Time Complexity: O(n²) in the average and worst cases. It is inefficient for large datasets due to its quadratic time complexity.

Quick Sort:  
  
Time Complexity: O(n log n) on average, making it much more efficient for large datasets compared to Bubble Sort. Its worst-case complexity is O(n²), but this can be mitigated with good pivot selection strategies (e.g., using random pivot or median-of-three).  
Why Quick Sort is Generally Preferred:  
  
Efficiency: Quick Sort is generally preferred due to its average-case time complexity of O(n log n), which is significantly better than Bubble Sort's O(n²).  
Practical Performance: Despite its worst-case time complexity, Quick Sort performs very well in practice due to its divide-and-conquer approach and efficient partitioning.  
In-place Sorting: Quick Sort sorts in place and requires minimal additional memory, unlike Merge Sort, which requires additional space for merging.

1. Analysis  
   Time Complexity Analysis:  
     
   Add Operation:  
     
   Best Case: O(1) (if there is space in the array)  
   Worst Case: O(n) (if resizing is needed, which involves copying elements to a new array)  
   Search Operation:  
     
   Best Case: O(1) (if the employee is at the first position)  
   Average Case: O(n) (linear search through the array)  
   Worst Case: O(n) (if the employee is at the end or not found)  
   Traverse Operation:  
     
   Time Complexity: O(n) (each employee needs to be accessed and printed)  
   Delete Operation:  
     
   Best Case: O(1) (if the employee to be deleted is at the end)  
   Worst Case: O(n) (if the employee is at the beginning or middle, and shifting elements is needed)
2. 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.
3. 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.
4. 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.