# DATA STRUCTURES AND ALGORITHM – E-COMMERCE PLATFORM SEARCH FUNCTION

# 1. Understanding Asymptotic Notation

## Big O Notation

Big O notation describes the upper bound of an algorithm's time or space complexity relative to the input size (n). It characterizes the worst-case growth rate, ignoring constants and lower-order terms. For example:

* O(1): Constant time (independent of n)
* O(log n): Logarithmic time (e.g., binary search)
* O(n): Linear time (e.g., linear search)
* O(n²): Quadratic time (e.g., nested loops)

## Search Operation Scenarios

Best Case: Minimum steps (e.g., target is the first element in linear search → O(1))

Average Case: Expected steps for random inputs (e.g., linear search averages O(n); binary search O(log n))

Worst Case: Maximum steps (e.g., target absent → linear search O(n), binary search O(log n))

# 2. Setup: Product Class (Java)

class Product {  
 int productId;  
 String productName;  
 String category;  
  
 public Product(int productId, String productName, String category) {  
 this.productId = productId;  
 this.productName = productName;  
 this.category = category;  
 }  
  
 @Override  
 public String toString() {  
 return "ID: " + productId + ", Name: '" + productName + "', Category: '" + category + "'";  
 }  
}

# 3. Implementation

## Linear Search (Unsorted Array)

Time Complexity: O(n)

public static Product linearSearch(List<Product> products, int targetId) {  
 for (Product product : products) {  
 if (product.productId == targetId) {  
 return product;  
 }  
 }  
 return null;  
}

## Binary Search (Sorted Array by productId)

Time Complexity: O(log n)

public static Product binarySearch(List<Product> products, int targetId) {  
 int low = 0, high = products.size() - 1;  
 while (low <= high) {  
 int mid = (low + high) / 2;  
 Product midProduct = products.get(mid);  
 if (midProduct.productId == targetId) {  
 return midProduct;  
 } else if (midProduct.productId < targetId) {  
 low = mid + 1;  
 } else {  
 high = mid - 1;  
 }  
 }  
 return null;  
}

# 4. Analysis

## Time Complexity Comparison

| Algorithm | Best Case | Average Case | Worst Case |  
|--------------------|--------------|-------------------|---------------|  
| Linear Search | O(1) | O(n) | O(n) |  
| Binary Search | O(1) | O(log n) | O(log n) |

## Suitability for E-commerce Platform

* - Binary Search is optimal for large datasets due to O(log n) complexity.  
  - Requirement: Products must be sorted (e.g., by productId).  
  - Linear Search is suitable for small/unsorted datasets but inefficient at scale (O(n)).

## Trade-offs

* - Binary Search: Requires sorted data (adds O(n log n) preprocessing). Ideal for static/rarely updated catalogs.
* - Linear Search: Simpler, no sorting needed. Use only if data changes frequently and size is small.

# Output Demonstration

List<Product> productsUnsorted = new ArrayList<>(Arrays.asList(  
 new Product(101, "Laptop", "Electronics"),  
 new Product(102, "Desk Chair", "Furniture"),  
 new Product(100, "Smartphone", "Electronics"),  
 new Product(103, "Coffee Mug", "Kitchen")  
));  
  
// Sort for binary search  
productsUnsorted.sort(Comparator.comparingInt(p -> p.productId));  
  
System.out.println("Linear Search (ID 100): " + linearSearch(productsUnsorted, 100));  
System.out.println("Binary Search (ID 100): " + binarySearch(productsUnsorted, 100));  
System.out.println("Binary Search (ID 999): " + binarySearch(productsUnsorted, 999));

## Output

Linear Search (ID 100): ID: 100, Name: 'Smartphone', Category: 'Electronics'  
Binary Search (ID 100): ID: 100, Name: 'Smartphone', Category: 'Electronics'  
Binary Search (ID 999): null

# Conclusion

For an e-commerce platform:

* Use Binary Search for large product catalogs to ensure fast searches.
* Maintain sorted order (e.g., during product updates).
* Avoid Linear Search except for trivial use cases.

Optimization Tip: For dynamic datasets, combine binary search with efficient sorting (e.g., use a TreeMap or custom comparator with sorted collections).