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

**1.**

Big O notation is a powerful tool used in computer science to describe the time complexity or space complexity of algorithms. Big-O is a way to express the upper bound of an algorithm’s time or space complexity.

* It describes the asymptotic behavior (order of growth of time or space in terms of input size) of a function, not its exact value.
* It can be used to compare the efficiency of different algorithms or data structures.
* It provides an upper limit on the time taken by an algorithm in terms of the size of the input. We mainly consider the worst-case scenario of the algorithm to find its time complexity in terms of Big O.
* It’s denoted as O(f(n)), where f(n) is a function that represents the number of operations (steps) that an algorithm performs to solve a problem of size n.

Big O notation helps in analyzing algorithms in the following ways:

1. Comparing Algorithm Efficiency:

This allows us to compare the efficiency of different algorithms for solving the same problem. By looking at the Big O notation of two algorithms, we can quickly determine which one will perform better for large input sizes.

2. Predicting Algorithm Behaviour:

Big O notation helps us predict how an algorithm will perform as the input data grows. This is crucial for understanding algorithms' scalability and ensuring they can efficiently handle larger datasets.

3. Optimizing Code:

Understanding the Big O complexity of an algorithm is essential for optimizing code. By identifying complex algorithms, developers can focus on improving those parts of the codebase to make their software more efficient.

4. Resource Management:

Big O notation is also relevant for resource management, especially in resource-constrained environments such as embedded systems or server environments. It helps developers make informed decisions about memory usage, processing power, and other resources.

5. Problem-Solving Approach:

When solving complex problems, knowing the Big O complexity of different algorithms can guide the selection of appropriate data structures and algorithms. This helps devise efficient solutions to real-world problems.

The best, average, and worst-case scenarios for search operations.

1. Worst Case Analysis (Mostly used)

* In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed.
* For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search()function compares it with all the elements of arr[] one by one.
* This is the most commonly used analysis of algorithms (We will be discussing below why). Most of the time we consider the case that causes maximum operations.

2. Best Case Analysis (Very Rarely used)

* In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed.
* For Linear Search, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So, the order of growth of time taken in terms of input size is constant.

3. Average Case Analysis (Rarely used)

* In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs.
* We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So, we sum all the cases and divide the sum by (n+1). We take (n+1) to consider the case when the element is not present.

Summary of all the best, average, and worst-case scenarios for search operations is given in the following table.

|  |  |  |
| --- | --- | --- |
| Case | Linear Search | Binary Search |
| Best | O(1) | O(1) |
| Average | O(n/2) ≈ O(n) | O(log n) |
| Worst | O(n) | O(log n) |

**2,3.**

**Code**

public class Product{

    int productId;

    String productName;

    String category;

    public Product(int id, String name, String category){

        this.productId=id;

        this.productName=name;

        this.category=category;

    }

}

public class Search{

    public static Product linearSearch(Product[] products, String name) {

        for (int i=0;i<products.length;i++) {

            if (products[i].productName.equalsIgnoreCase(name)) {

                return products[i];

            }

        }

        return null;

    }

public static Product binarySearch(Product[] products, String name) {

        int low=0, high=products.length - 1;

        while (low<=high) {

            int mid = low + (high-low)/2;

            int cmp = products[mid].productName.compareToIgnoreCase(name);

            if (cmp == 0) {

                return products[mid];

            } else if (cmp<0) {

                low = mid + 1;

            } else {

                high = mid - 1;

            }

        }

        return null;

    }

}

public class ProductTest {

    public static void main(String[] args) {

        Product[] products = {

            new Product(1,"Keyboard", "Electronics"),

            new Product(2, "Laptop", "Electronics"),

            new Product(3,"Shoes", "Footwear"),

            new Product(4, "Book", "Stationery")

        };

        Product linearResult=Search.linearSearch(products,"keyboard");

        System.out.println("id: "+linearResult.productId+"\nCategory: "+linearResult.category);

        Product[] sorted\_products = {

            new Product(1, "Book", "Stationery"),

            new Product(2, "Keyboard", "Electronics"),

            new Product(3, "Laptop", "Electronics"),

            new Product(4, "Shoes", "Footwear")

        };

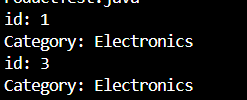
        Product binaryResult=Search.binarySearch(sorted\_products,"laptop");

        System.out.println("id: "+binaryResult.productId+"\nCategory: "+binaryResult.category);

    }

}

**Output**

****

**4.**

The time complexity of linear search is O(n), where n is the number of elements in the array. In the worst-case scenario, the algorithm has to check all elements before finding the target value, making it inefficient for large datasets.

The time complexity of binary search is O(log n), where n is the number of elements in the array. With each comparison, the search space is reduced by half, making it a highly efficient algorithm for searching large datasets.

Time complexity of Linear Search – O(n)

Time complexity of Binary Search – O(log n)

**Differences Between Binary Search and Linear Search**

1. **Type of input arrays**:

Binary search requires the input array to be sorted, whereas linear search can work on both sorted and unsorted arrays.

1. **Efficiency**:

Binary search is more efficient than linear search, especially for large datasets. Binary search has a time complexity of O(log n), while linear search has a time complexity of O(n).

1. **Algorithm Complexity**:

Linear search is a simpler algorithm compared to binary search, making it easier to understand and implement.

1. **Number of Comparisons**:

Linear search may require up to n comparisons (where n is the number of elements in the array), while binary search requires at most log2(n+1) comparisons.

**Binary search is better for performance, but only if the product list is sorted. For dynamic/unsorted data or very small datasets, linear search is simpler.**

**Exercise 7: Financial Forecasting**

**1,2,3.**

**Code**

public class FinancialForecast {

    public static double calcFutureValue(double principal, double rate, int years){

        if(years==0){

            return principal;

        }

        return calcFutureValue(principal\*(1+rate), rate, years-1);

    }

    public static void main(String[] args) {

        double principal=15000.00;

        double rate=0.05;

        int years=10;

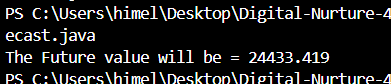
        double futureValue=calcFutureValue(principal, rate, years);

        System.out.printf("The Future value will be = %.3f",futureValue);

    }

}

**Output**



**4.**

Time Complexity of the recursive algorithm is O(n).

For each year, we perform one recursive call. So, for n years, n recursive calls.

Recursion can lead to stack overflow, if n is large.

We can use tail recursion or iterative approach or memoization for optimized results.

**Exercise 1: Inventory Management System**

**1.**

In a large warehouse, there can be thousands of products.

Efficient storage (using optimal data structures) ensures fast retrieval and updates.

Algorithms help maintain performance, especially when handling search, insert, update, or delete operations frequently.

Suitable Data Structures:

HashMap (Java):

Best for fast lookups using productId as key.

Ideal for add/update/delete/search in O(1) average time.

ArrayList / LinkedList:

Simpler but slower (especially for search/delete, which are O(n)).

**2.**

**Code**

class Product {

    int productId;

    String productName;

    int quantity;

    double price;

    public Product(int productId, String productName, int quantity, double price) {

        this.productId = productId;

        this.productName = productName;

        this.quantity = quantity;

        this.price = price;

    }

    @Override

    public String toString() {

        return productId + " | " + productName + " | " + quantity + " | $" + price;

    }

}

import java.util.HashMap;

class InventoryManager {

    private HashMap<Integer, Product> inventory = new HashMap<>();

    public void addProduct(Product p) {

        inventory.put(p.productId, p);

    }

    public void updateProduct(int id, String name, int quantity, double price) {

        if (inventory.containsKey(id)) {

            Product p = inventory.get(id);

            p.productName = name;

            p.quantity = quantity;

            p.price = price;

        }

    }

    public void deleteProduct(int id) {

        inventory.remove(id);

    }

    public void displayInventory() {

        for (Product p : inventory.values()) {

            System.out.println(p);

        }

    }

}

**3.**

| **Operation** | **Time Complexity (HashMap)** | **Reason** |
| --- | --- | --- |
| Add | O(1) | Direct key-based insertion |
| Update | O(1) | Direct key access/update |
| Delete | O(1) | Key-based removal |
| Search/Display | O(n) | Loop over all products |

If requirement of sorting (e.g., by price) is there, one can consider using a **TreeMap** or sort the values manually. One can use a **database** in real applications to handle persistence and concurrent access and add validation and error handling in a production-level system.

**Exercise 3: Sorting Customer Orders**

**1.**

**Bubble Sort**

* Compares adjacent items and swaps them if they are in the wrong order.
* Repeats this process until the list is sorted.
* Best case time complexity: O(n)
* Average case time complexity: O(n²)
* Worst case time complexity: O(n²)

**Insertion Sort**

* Builds the sorted list one item at a time.
* Each new item is compared with already sorted items and placed in the correct position.
* Best case time complexity: O(n)
* Average case time complexity: O(n²)
* Worst case time complexity: O(n²)

**Quick Sort**

* A **divide-and-conquer** algorithm.
* Picks a pivot and partitions the array into elements smaller and greater than the pivot.
* Recursively applies the same logic to the subarrays.
* Best case time complexity: O(n log n)
* Average case time complexity: O(n log n)
* Worst case time complexity: O(n²) (rare; depends on pivot choice)

**Merge Sort**

* Also divide-and-conquer.
* Divides array into halves, sorts each recursively, and merges them.

**Time Complexity:**

* Best case time complexity: O(n log n)
* Average case time complexity: O(n log n)
* Worst case time complexity: O(n log n)

**2,3.**

**Code**

class Order {

    int orderId;

    String customerName;

    double totalPrice;

    public Order(int orderId, String customerName, double totalPrice) {

        this.orderId = orderId;

        this.customerName = customerName;

        this.totalPrice = totalPrice;

    }

    @Override

    public String toString() {

        return orderId + " | " + customerName + " | $" + totalPrice;

    }

}

public static void bubbleSort(Order[] orders) {

    int n = orders.length;

    for (int i = 0; i < n - 1; i++) {

        boolean swapped = false;

        for (int j = 0; j < n - i - 1; j++) {

            if (orders[j].totalPrice > orders[j + 1].totalPrice) {

                Order temp = orders[j];

                orders[j] = orders[j + 1];

                orders[j + 1] = temp;

                swapped = true;

            }

        }

        if (!swapped) break;

    }

}

public static void quickSort(Order[] orders, int low, int high) {

    if (low < high) {

        int pi = partition(orders, low, high);

        quickSort(orders, low, pi - 1);

        quickSort(orders, pi + 1, high);

    }

}

private static int partition(Order[] orders, int low, int high) {

    double pivot = orders[high].totalPrice;

    int i = low - 1;

    for (int j = low; j < high; j++) {

        if (orders[j].totalPrice < pivot) {

            i++;

            Order temp = orders[i];

            orders[i] = orders[j];

            orders[j] = temp;

        }

    }

    Order temp = orders[i + 1];

    orders[i + 1] = orders[high];

    orders[high] = temp;

    return i + 1;

}

**4.**

|  |  |  |
| --- | --- | --- |
| Criteria | Bubble Sort | Quick Sort |
| Time Complexity (Average case) | O(n²) | O(n log n) |
| Time Complexity (Worst case) | O(n²) | O(n²) |
| Time Complexity (Best case) | O(n) | O(n log n) |
| Space Complexity | O(1) | O(log n) |

Bubble Sort becomes inefficient as the dataset becomes large, whereas, Quick Sort, although recursive, performs very well on average and is widely used in production systems.

**Exercise 4: Employee Management System**

**1.**

Arrays are contiguous blocks of memory.

Any element in an array can be accessed directly using its index in O(1) time ie.. arrays support random access.

Each element of an array is stored at a fixed offset from the base address: Address of arr[i] = base + (i × size of each element).

Advantages of Arrays

* Fast random access
* Simple and easy to use
* Cache-friendly due to contiguous memory layout
* Memory-efficient for fixed-size datasets

**2,3.**

**Code**

class Employee {

    int employeeId;

    String name;

    String position;

    double salary;

    public Employee(int employeeId, String name, String position, double salary) {

        this.employeeId = employeeId;

        this.name = name;

        this.position = position;

        this.salary = salary;

    }

    @Override

    public String toString() {

        return employeeId + " | " + name + " | " + position + " | $" + salary;

    }

}

public class EmployeeManager {

    private Employee[] employees;

    private int count;

    public EmployeeManager(int size) {

        employees = new Employee[size];

        count = 0;

    }

    public void addEmployee(Employee emp) {

        if (count < employees.length) {

            employees[count++] = emp;

        } else {

            System.out.println("Array full. Cannot add more employees.");

        }

    }

    public Employee searchEmployee(int empId) {

        for (int i = 0; i < count; i++) {

            if (employees[i].employeeId == empId) {

                return employees[i];

            }

        }

        return null;

    }

    public void displayAll() {

        for (int i = 0; i < count; i++) {

            System.out.println(employees[i]);

        }

    }

    public void deleteEmployee(int empId) {

        for (int i = 0; i < count; i++) {

            if (employees[i].employeeId == empId) {

                for (int j = i; j < count - 1; j++) {

                    employees[j] = employees[j + 1];

                }

                employees[--count] = null;

                System.out.println("Employee deleted.");

                return;

            }

        }

        System.out.println("Employee not found.");

    }

**4.**

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Explanation |
| Add | O(1) | Append at the end using index |
| Search | O(n) | Linear search through array |
| Traverse | O(n) | Visit each element |
| Delete | O(n) | Find and shift elements after deletion |

Limitations of Arrays:

* Fixed size: Cannot grow/shrink dynamically
* Insertion/Deletion is costly (O(n)) due to shifting elements
* Wastes memory if the array is underutilized

When to Use Arrays:

* When the number of elements is known and fixed
* When fast random access is needed
* When working with small, static datasets

**Exercise 5: Task Management System**

**1.**

Singly Linked List

* A linear data structure where each node points to the next node.
* Structure: Data -> Next
* Operations: traversal is one-way only (forward).

Doubly Linked List

* Each node points to both the next and the previous node.
* Structure: Prev <- Data -> Next
* Allows forward and backward traversal.
* Slightly more memory-intensive than singly linked lists.

**2,3.**

**Code**

class Task {

    int taskId;

    String taskName;

    String status;

    public Task(int taskId, String taskName, String status) {

        this.taskId = taskId;

        this.taskName = taskName;

        this.status = status;

    }

    @Override

    public String toString() {

        return taskId + " | " + taskName + " | " + status;

    }

}

class Node {

    Task task;

    Node next;

    public Node(Task task) {

        this.task = task;

        this.next = null;

    }

}

public class TaskManager {

    private Node head;

    public void addTask(Task task) {

        Node newNode = new Node(task);

        if (head == null) {

            head = newNode;

        } else {

            Node current = head;

            while (current.next != null)

                current = current.next;

            current.next = newNode;

        }

    }

    public Task searchTask(int taskId) {

        Node current = head;

        while (current != null) {

            if (current.task.taskId == taskId) {

                return current.task;

            }

            current = current.next;

        }

        return null;

    }

    public void displayTasks() {

        Node current = head;

        while (current != null) {

            System.out.println(current.task);

            current = current.next;

        }

    }

    public void deleteTask(int taskId) {

        if (head == null) return;

        if (head.task.taskId == taskId) {

            head = head.next;

            return;

        }

        Node current = head;

        while (current.next != null) {

            if (current.next.task.taskId == taskId) {

                current.next = current.next.next;

                return;

            }

            current = current.next;

        }

        System.out.println("Task ID not found.");

    }

}

**4.**

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Explanation |
| Add | O(n) | Traverse to the end |
| Search | O(n) | Linear search |
| Traverse | O(n) | One pass through the list |
| Delete | O(n) | Find and adjust pointers |

|  |  |  |
| --- | --- | --- |
| Feature | Array | Linked List |
| Memory Allocation | Fixed, contiguous | Dynamic, scattered |
| Insertion/Deletion | Costly (shift elements) | Efficient (adjust pointers) |
| Access Time | O(1) random access | O(n) traversal required |
| Memory Efficiency | Less (no extra pointers) | More (each node has a pointer) |
| Ideal For | Static datasets | Dynamic data with frequent add/delete |

**Exercise 6: Library Management System**

**1.**

Linear Search

* Checks each element one by one.
* Works on unsorted data.
* Best case time complexity: O(1) (match at start)
* Average case time complexity: O(n)
* Worst case time complexity: O(n) (match at end or not found)

Binary Search

* Works only on sorted data.
* Divides the array in halves repeatedly.
* Best case time complexity:: O(1)
* Average case time complexity: O(log n)
* Worst case time complexity: O(log n)

**2,3.**

**Code**

import java.util.List;

class Book {

    int bookId;

    String title;

    String author;

    public Book(int bookId, String title, String author) {

        this.bookId = bookId;

        this.title = title;

        this.author = author;

    }

    @Override

    public String toString() {

        return bookId + " | " + title + " | " + author;

    }

    public static Book linearSearchByTitle(List<Book> books, String targetTitle) {

        for (Book book : books) {

            if (book.title.equalsIgnoreCase(targetTitle)) {

            return book;

            }

        }

    return null;

    }

    public static Book binarySearchByTitle(List<Book> books, String targetTitle) {

        int left = 0, right = books.size() - 1;

        while (left <= right) {

            int mid = (left + right) / 2;

            int cmp = books.get(mid).title.compareToIgnoreCase(targetTitle);

            if (cmp == 0)

                return books.get(mid);

            else if (cmp < 0)

                left = mid + 1;

            else

                right = mid - 1;

        }

        return null;

    }

}

**4.**

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

When to Use Linear Search:

* Unsorted or small datasets
* Quick implementation
* Dynamic data with frequent insertions

When to Use Binary Search

* Large datasets that are sorted
* Performance-critical apps
* Static or rarely modified lists