**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")

        };

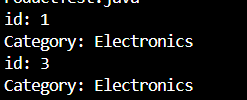
        Product binaryResult=Search.binarySearch(sorted\_products,"laptop");

        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;

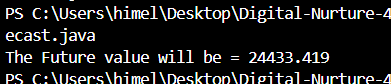
        double futureValue=calcFutureValue(principal, rate, years);

        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.