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

Big O notation is a mathematical concept used in computer science to describe the efficiency and scalability of algorithms. It focuses primarily on how an algorithm's performance changes in response to changes in the size of the input, rather than the exact time it takes to execute. This helps developers and computer scientists evaluate, compare, and choose algorithms that perform better in real-world scenarios, especially when working with large data sets.

**Why is Big O Notation Important?**

When writing code, especially for large-scale applications like search engines, e-commerce platforms, or financial tools, it's not enough for a solution to work — it must work efficiently.

Imagine you have an algorithm that works great for 10 items but becomes painfully slow with 10 million items. Big O helps you predict and avoid such bottlenecks.

Big O notation answers these questions:

* Will my algorithm scale well as input grows?
* How does performance degrade with more data?
* Is there a faster or more memory-efficient alternative?

**How Big O Works?**

Big O doesn't measure actual time in seconds or milliseconds. Instead, it expresses performance as a **function of input size n**:

n = size of the input (e.g., number of elements in an array).

Then, it estimates how **many steps** or **operations** an algorithm takes in relation to n.

**Best, Average, and Worst Case:**

Big O focuses on the worst-case scenario, which is important for reliability.

But algorithms can also be analysed in terms of:

* **Best Case**: Minimum steps (e.g., item is at the first index).
* **Average Case**: Expected steps over many inputs.
* **Worst Case**: Maximum steps (e.g., item is not present).

**Example**: Linear search

* Best case: O(1) → first item matches
* Average case: O(n/2) → middle item
* Worst case: O(n) → last item or not found

**Big O simplifies O(n/2)** to **O(n)** — because constants are dropped for generality.

**How It Helps in Practice?**

1. **Choosing Efficient Algorithms**

For sorting, prefer Merge Sort (O(n log n)) over Bubble Sort (O(n²)) for large datasets.

1. **Improving Code**

Rewriting nested loops into better structures can significantly reduce execution time.

1. **Interview Preparation**

Technical interviews often focus on Big O to test your algorithmic thinking.

1. **Optimizing Systems**

In real systems (e.g., Google search, Amazon checkout), performance is critical. Big O helps you build systems that scale efficiently.

**Linear Search**

**Definition**: A simple search where each element is checked **one-by-one**, from start to end.

**Best Case:**

* **Scenario**: The element is found at the **first position**.
* **Time Complexity**: O(1) (constant time)

**Eg**:

Array: [10, 20, 30, 40, 50]

Target: 10 (at index 0)

**Average Case:**

* **Scenario**: The element is somewhere in the **middle** of the array.
* **Time Complexity**: O(n/2) → simplified to O(n)
* On average, you'd check about **half** the elements.

**Eg:**

Array: [10, 20, 30, 40, 50]

Target: 30 (at index 2 of 5)

**Worst Case:**

* **Scenario 1**: The element is at the **last index**.
* **Scenario 2**: The element **does not exist**.
* **Time Complexity**: O(n)

Array: [10, 20, 30, 40, 50]

Target: 90 (not present)

**Binary Search**

**Definition**: A fast search technique for **sorted arrays**, where the middle element is compared and half the array is eliminated in each step.

**Best Case:**

* **Scenario**: The element is at the **middle** index in the first comparison.
* **Time Complexity**: O(1)

**Eg:**

Sorted Array: [10, 20, 30, 40, 50]

Target: 30 (middle element)

**Average Case:**

* **Scenario**: The element is not in the middle but somewhere in the array.
* **Time Complexity**: O(log n)
* Each step halves the search space.

**Eg:**

Sorted Array: [10, 20, 30, 40, 50, 60, 70, 80]

Target: 70 → checked after 3 steps

**Worst Case:**

* **Scenario**: The element is not present or is found after **log₂(n)** splits.
* **Time Complexity**: O(log n)

**Eg**:

Sorted Array: [10, 20, 30, 40, 50, 60]

Target: 15 (not present)

**Code:**

public **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 "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

public **class LinearSearch** {

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

for (Product p : products) {

if (p.productName.equalsIgnoreCase(name)) {

return p;

}

}

return null;

} }

import java.util.Arrays;

import java.util.Comparator;

public **class BinarySearch** {

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

Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

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

if (cmp == 0) return products[mid];

else if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

}

public **class SearchTest** {

public static void main(String[] args) {

Product[] products = {

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

new Product(2, "Shoes", "Fashion"),

new Product(3, "Watch", "Accessories"),

new Product(4, "Mobile", "Electronics")

};

System.out.println("Linear Search Result:");

Product result1 = LinearSearch.searchByName(products, "Watch");

System.out.println(result1 != null ? result1 : "Product not found");

System.out.println("\nBinary Search Result:");

Product result2 = BinarySearch.searchByName(products, "Watch");

System.out.println(result2 != null ? result2 : "Product not found");

}

}

**Output:**

**Linear Search Result:**

**Product ID: 3, Name: Watch, Category: Accessories**

**Binary Search Result:**

**Product ID: 3, Name: Watch, Category: Accessories**

**Compare the Time Complexity of Linear and Binary Search Algorithms**

| Aspect | Linear Search | Binary Search |
| --- | --- | --- |
| Best Case | O(1) | O(1) |
| Average Case | O(n) | O(log n) |
| Worst Case | O(n) | O(log n) |
| Requirement | Works on any array | Requires sorted array |
| Space Complexity | O(1) | O(1) (iterative version) |

Explanation:

* Linear Search scans each element in the array one by one until it finds the target or reaches the end. Hence, the time complexity grows linearly with input size.
* Binary Search repeatedly divides the sorted array in half, reducing the search space by 50% in each step. Thus, it has logarithmic time complexity.

For example, if you have:

* 1000 items → Linear search may take up to 1000 steps, binary search only 10 steps.
* 1,000,000 items → Linear: 1,000,000, Binary: ~20 steps

**Which Algorithm is More Suitable for Your Platform and Why?**

Binary Search is more suitable, especially for an e-commerce platform like:

* Amazon, Flipkart, or Myntra where:
  + There are millions of products.
  + Users expect instant search results.
  + Data can be pre-sorted (e.g., by name, ID, category).

Advantages of Binary Search for E-commerce:

* Fast performance even for large datasets.
* Scalable for real-time user queries.
* Can be used efficiently with backend databases or in-memory indexes.

When to Consider Linear Search:

* When the product list is small or unsorted.
* When you're performing a one-time search on a small dataset.
* For ad-hoc searches during testing or admin utilities.

**Exercise 7: Financial Forecasting**

Recursion is a method in algorithms where a function calls itself to solve a problem by dividing it into smaller sub-problems of the same type. It is widely used in algorithm design and data structure traversal, especially where problems exhibit self-similarity or follow a divide-and-conquer approach.

In data structures, recursion is particularly useful for processing hierarchical or non-linear structures like trees and graphs.

**How Recursion Works?**

A recursive algorithm typically has:

1. Base Case – The simplest version of the problem that can be solved directly.
2. Recursive Case – The part that reduces the problem into a smaller instance and calls the function again.

Example: Recursive Algorithm for Factorial

int factorial(int n) {

if (n == 0) return 1;

return n \* factorial(n - 1);

}

**Recursion in Data Structures**

**Trees**

* Tree structures are naturally recursive: each node has subtrees (left and right).
* Recursive Traversals:
  + Inorder (Left, Root, Right)
  + Preorder (Root, Left, Right)
  + Postorder (Left, Right, Root)

void inorder(TreeNode node) {

if (node == null) return;

inorder(node.left);

System.out.print(node.value + " ");

inorder(node.right);

}

**Linked Lists**

* Operations like reverse, search, and traversal can be written recursively.

void printList(Node head) {

if (head == null) return;

System.out.println(head.data);

printList(head.next);

}

**Recursion in Algorithm Design**

**Divide and Conquer Algorithms**

* Problems are split into smaller sub-problems, solved recursively, and merged.

Examples:

| Algorithm | Structure | How Recursion Simplifies |
| --- | --- | --- |
| Merge Sort | Arrays | Splits the array, sorts each half recursively, then merges |
| Quick Sort | Arrays | Picks a pivot, recursively sorts partitions |
| Binary Search | Arrays (sorted) | Searches in one half recursively |

int binarySearch(int[] arr, int low, int high, int target) {

if (low > high) return -1;

int mid = (low + high) / 2;

if (arr[mid] == target) return mid;

else if (target < arr[mid]) return binarySearch(arr, low, mid - 1, target);

else return binarySearch(arr, mid + 1, high, target);

}

**Graphs**

* Depth-First Search (DFS) is naturally recursive as it explores nodes by diving deep into each path.

void dfs(int node, boolean[] visited, List<List<Integer>> graph) {

visited[node] = true;

for (int neighbor : graph.get(node)) {

if (!visited[neighbor]) dfs(neighbor, visited, graph);

} }

**Backtracking Algorithms**

* Solve problems by exploring all possibilities recursively.
* Examples: N-Queens, Sudoku solver, Permutations/Combinations.

void generatePermutations(List<Integer> list, int index) {

if (index == list.size()) {

System.out.println(list);

return;

}

for (int i = index; i < list.size(); i++) {

Collections.swap(list, i, index);

generatePermutations(list, index + 1);

Collections.swap(list, i, index);

}

}

**Time and Space Considerations**

Pros:

* Elegant solutions for recursive structures.
* Readable and concise code for complex problems.

Cons:

* May cause stack overflow for deep recursion.
* Overlapping subproblems (e.g., Fibonacci) lead to redundant calls.

Optimization:

* Use memoization (top-down dynamic programming) to store results of sub-problems.
* Use tail recursion or convert to iteration to reduce stack usage.

| Aspect | Recursion Benefit |
| --- | --- |
| Algorithm Design | Simplifies divide-and-conquer & backtracking |
| Data Structure Traversal | Matches the recursive nature of trees/graphs |
| Code Clarity | Cleaner than loops for nested/recursive logic |
| Performance | Can be optimized using memoization or DP |

In conclusion, recursion simplifies algorithm design by breaking problems into smaller versions of themselves, especially in trees, graphs, sorting, and searching. While powerful, recursive solutions should be optimized to avoid excessive memory and computation.

**Code:**

public **class FinancialForecasting** {

public static double calculateFutureValue(double presentValue, double rate, int years) {

if (years == 0) {

return presentValue;

}

return calculateFutureValue(presentValue, rate, years - 1) \* (1 + rate);

}

}

public **class ForecastTest** {

public static void main(String[] args) {

double presentValue = 10000.0;

double growthRate = 0.08;

int years = 5;

double futureValue = FinancialForecasting.calculateFutureValue(presentValue, growthRate, years);

System.out.printf("Future Value after %d years: ₹%.2f%n", years, futureValue);

}

}

**Output:**

**Future Value after 5 years: ₹14693.28**

**Time Complexity: O(n)**

* The function is called once per year, decrementing years by 1 each time until it reaches 0.
* So, for n years, there are exactly n recursive calls.

Efficient for small-to-medium n, but scales linearly.

**Space Complexity: O(n)**

* Each recursive call is stored in the call stack.
* So for n years, the stack depth is n.
* This could lead to StackOverflowError if n is very large (e.g., 10,000+ years — unrealistic for most forecasting, but important to consider).

**Optimization Strategies**

**1. Convert to Iteration**

The most common and effective optimization is to **replace recursion with iteration**.

In iterative solutions:

* A loop performs the same computation as the recursive calls.
* There's no stack overhead, making the solution more **memory-efficient**.
* The space complexity is reduced to **O(1)**.

**Why it works**:  
In problems like compound growth (e.g., future value calculations), each step depends only on the previous result. This sequential dependency can be easily handled using a loop.

**2. Memoization (Caching)**

For more complex recursive problems — such as those involving **variable annual growth rates**, **multiple investment paths**, or **non-linear dependencies** — **memoization** can optimize performance.

**Memoization** is a technique where the results of expensive function calls are **stored** and **reused** when the same inputs occur again.

* This prevents repeated calculations for the same inputs.
* It reduces redundant recursive calls.
* Time complexity remains **O(n)**, but **actual execution time improves** significantly.

Memoization is especially useful in problems with **overlapping subproblems**, such as Fibonacci sequences or decision trees in financial modeling.

**3. Tail Recursion (Language-Specific)**

In languages that support **tail call optimization (TCO)**, transforming a recursive method into a **tail-recursive** one allows the compiler or interpreter to **reuse stack frames**, thus avoiding stack overflow.

* Tail recursion means the recursive call is the **last** operation in the function.
* However, **Java does not support TCO**, so this optimization is not applicable in standard Java environments.