**Exercise 1: Inventory Management System**

**STEP 1 :**

**o Explain why data structures and algorithms are essential in handling large inventories.**

* When handling large inventories, efficiency in data storage and retrieval is crucial. The operations you typically perform on inventory data include:
* Adding new products: You need to quickly insert new items.
* Updating product details: You need to find and modify items efficiently.
* Deleting products: You need to quickly remove items from your system.
* Searching for products: You need fast access to find items by their attributes.

Without the right data structures and algorithms, these operations could become slow and resource-intensive, especially as the inventory grows. Efficient data structures help maintain performance and ensure that operations are executed in a timely manner.

**o Discuss the types of data structures suitable for this problem.**

 ArrayList: Allows fast access to elements but is less efficient for adding and removing items in the middle of the list.

 HashMap: Provides average O(1) time complexity for add, update, and delete operations using a key-value pair. This is generally a good choice for this problem because you can use the product ID as the key for quick access.

 TreeMap: Similar to HashMap but maintains a sorted order of the keys. Useful if you need to frequently access elements in a sorted manner.

 LinkedList: Good for scenarios where you need efficient insertions and deletions but not necessarily fast random access.

**STEP 2 & 3 : Setup and Implementation**

Define the Product Class

public class Product {

private String productId;

private String productName;

private int quantity;

private double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public String getProductId() { return productId; }

public void setProductId(String productId) { this.productId = productId; }

public String getProductName() { return productName; }

public void setProductName(String productName) { this.productName = productName; }

public int getQuantity() { return quantity; }

public void setQuantity(int quantity) { this.quantity = quantity; }

public double getPrice() { return price; }

public void setPrice(double price) { this.price = price; }

}

**For storing products, a HashMap is a suitable choice because it allows fast lookups by productId.**

import java.util.HashMap;

import java.util.Map;

public class InventorySystem {

private Map<String, Product> inventory;

public InventorySystem() {

this.inventory = new HashMap<>();

}

public void addProduct(Product product) {

inventory.put(product.getProductId(), product);

}

public void updateProduct(String productId, Product updatedProduct) {

if (inventory.containsKey(productId)) {

inventory.put(productId, updatedProduct);

}

}

public void deleteProduct(String productId) {

inventory.remove(productId);

}

public Product getProduct(String productId) {

return inventory.get(productId);

}

}

**STEP 4 :**

**o Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.**

 Add Product: O(1) on average. Inserting a product into a HashMap involves calculating the hash and placing it into the appropriate bucket.

 Update Product: O(1) on average. The HashMap lookup for the key is O(1), and the update operation is also O(1).

 Delete Product: O(1) on average. Removing a product involves a similar hash calculation and bucket access.

**o Discuss how you can optimize these operations.**

 Load Factor and Capacity: Adjust the HashMap's initial capacity and load factor based on the expected size of the inventory. A high load factor can degrade performance, while a low one may waste memory.

 Handling Collisions: Ensure that the hash function distributes entries well to avoid performance degradation due to collisions.

 Concurrency: If multiple threads might access the inventory concurrently, consider using ConcurrentHashMap or synchronizing access to ensure thread safety.

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

**STEP 1 :**

**o Explain Big O notation and how it helps in analyzing algorithms.**

* Big O notation is used to describe the time complexity of an algorithm, which tells us how the running time of an algorithm grows with the size of the input. It provides an upper bound on the time complexity, giving us a way to understand the worst-case scenario.
* O(1): Constant time. The algorithm takes the same amount of time regardless of input size.
* O(n): Linear time. The time grows linearly with the input size.
* O(log n): Logarithmic time. The time grows logarithmically with the input size.
* O(n^2): Quadratic time. The time grows quadratically with the input size.

**o Describe the best, average, and worst-case scenarios for search operations.**

* Linear Search:*

* Best Case: O(1) — The element is found at the first position.
* Average Case: O(n) — On average, you may have to check half the elements.
* Worst Case: O(n) — The element is not found, or is the last element.

* Binary Search (requires a sorted array):*

* Best Case: O(1) — The element is found at the middle of the array.
* Average Case: O(log n) — The search space is halved with each step.
* Worst Case: O(log n) — The search space reduces logarithmically until the element is found or the space is exhausted.

**STEP 2 : SETUP**

public class Product {

private String productId;

private String productName;

private String category;

public Product(String productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public String getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

public String toString() {

return "Product{" +

"productId='" + productId + '\'' +

", productName='" + productName + '\'' +

", category='" + category + '\'' +

'}';

}

}

**STEP 3: IMPLEMENTATION**

*Linear Search*

public class SearchAlgorithms {

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

for (Product product : products) {

if (product.getProductId().equals(productId)) {

return product;

}

}

return null;

}

}

*Binary Search*

import java.util.Arrays;

public class SearchAlgorithms\_binary {

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

Product midProduct = products[mid];

int cmp = midProduct.getProductId().compareTo(productId);

if (cmp == 0) {

return midProduct;

} else if (cmp < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public static void sortProductsById(Product[] products) {

Arrays.sort(products, (p1, p2) -> p1.getProductId().compareTo(p2.getProductId()));

}

}

**STEP 4 :**

**o Compare the time complexity of linear search and binary search algorithms.**

Time Complexity Comparison

Linear Search:

* + Time Complexity: O(n)
  + Scenario: Simple to implement, no requirement for sorted data.
  + Usage: Suitable for small datasets or unsorted data.

Binary Search:

* + Time Complexity: O(log n)
  + Scenario: Requires sorted data but is much faster for large datasets.
  + Usage: Ideal for large datasets where sorting is feasible and efficient.

**o Discuss which algorithm is more suitable for your platform and why.**

 **For Large Datasets**: **Binary Search** is more suitable as it significantly reduces search time compared to linear search. However, it requires the data to be sorted, which might involve an initial sorting cost (O(n log n)).

 **For Small Datasets or Unsorted Data**: **Linear Search** is simpler to implement and does not require sorted data. It is often used when the overhead of sorting data is not justified.

**Exercise 3: Sorting Customer Orders**

**STEP 1 :**

**o Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).**

* *Bubble Sort*

Description: Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

Time Complexity:

* Best Case: O(n) — when the list is already sorted.
* Average Case: O(n²) — typically involves nested loops.
* Worst Case: O(n²) — when the list is sorted in reverse order.

Pros:

* Simple to implement and understand.
* Useful for small datasets or educational purposes.

Cons:

* Inefficient for large datasets due to its O(n²) time complexity.
* *Insertion Sort*

Description: Insertion Sort builds the final sorted array one item at a time by repeatedly picking the next item and inserting it into the correct position.

Time Complexity:

* Best Case: O(n) — when the list is already sorted.
* Average Case: O(n²) — similar to Bubble Sort.
* Worst Case: O(n²) — when the list is sorted in reverse order.

Pros:

* More efficient than Bubble Sort for small or nearly sorted datasets.

Cons:

* Still O(n²) time complexity for large datasets.
* *Quick Sort*

Description: Quick Sort is a divide-and-conquer algorithm that selects a 'pivot' element, partitions the array into elements less than and greater than the pivot, and recursively sorts the partitions.

Time Complexity:

* Best Case: O(n log n) — when the pivot divides the list evenly.
* Average Case: O(n log n) — typical case.
* Worst Case: O(n²) — occurs when the smallest or largest element is consistently chosen as the pivot.

Pros:

* Generally faster than Bubble Sort and Insertion Sort for large datasets.
* Efficient with O(n log n) average-case time complexity.

Cons:

* Recursive and has a worst-case time complexity of O(n²) if not implemented with a good pivot strategy.
* *Merge Sort*

Description: Merge Sort is a divide-and-conquer algorithm that splits the array into halves, recursively sorts each half, and then merges the sorted halves.

Time Complexity:

* Best Case: O(n log n) — same for all cases.
* Average Case: O(n log n).
* Worst Case: O(n log n).

Pros:

* Consistent O(n log n) performance.
* Stable sort (preserves the order of equal elements).

Cons:

* Requires additional space proportional to the array size.

**STEP 2 :**

**o Create a class Order with attributes like orderId, customerName, and totalPrice**

public class Order {

private String orderId;

private String customerName;

private double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public String getOrderId() {

return orderId;

}

public String getCustomerName() {

return customerName;

}

public double getTotalPrice() {

return totalPrice;

}

public String toString() {

return "Order{" +

"orderId='" + orderId + '\'' +

", customerName='" + customerName + '\'' +

", totalPrice=" + totalPrice +

'}';

}

}

**STEP 3 :**

*Bubble Sort Implementation*

public class SortOrders\_bubble {

public static void bubbleSort(Order[] orders) {

int n = orders.length;

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

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

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

// Swap orders[j] and orders[j + 1]

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

}

*Quick Sort Implementation*

public class SortOrders\_quick {

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

if (low < high) {

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

quickSort(orders, low, pi - 1); // Before pi

quickSort(orders, pi + 1, high); // After pi

}

}

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

double pivot = orders[high].getTotalPrice();

int i = (low - 1); // Index of smaller element

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

if (orders[j].getTotalPrice() <= pivot) {

i++;

// Swap orders[i] and orders[j]

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;

}

}

**STEP 4 :**

**o Compare the performance (time complexity) of bubble sort and quick sort.**

* Performance Comparison

Bubble Sort:

* + Time Complexity: O(n²) in the average and worst cases.
  + Performance: Poor for large datasets due to its quadratic time complexity.

Quick Sort:

* + Time Complexity: O(n log n) on average, O(n²) in the worst case.
  + Performance: Much faster for large datasets in practice due to its average-case time complexity of O(n log n). Good implementations use techniques like randomized pivots or median-of-three to improve performance.

**o Discuss why Quick Sort is generally preferred over Bubble Sort.**

 Efficiency: Quick Sort is faster for large datasets compared to Bubble Sort due to its O(n log n) average-case time complexity.

 Divide and Conquer: Quick Sort's divide-and-conquer approach efficiently reduces the problem size in each recursive step.

 Practical Performance: Despite its worst-case O(n²) complexity, Quick Sort's average performance and practical speed often make it the preferred choice for sorting tasks.

**Exercise 4: Employee Management System**

**STEP 1 :**

**o Explain how arrays are represented in memory and their advantages**

* Memory Layout:
* Arrays are stored in contiguous memory locations, meaning that each element in the array is placed next to the previous one.
* The array's base address is the memory location of the first element.
* Accessing elements is efficient because you can compute the memory address of any element using a formula involving the base address and the index.

Advantages of Arrays:

* Direct Access: Arrays allow constant-time O(1) access to elements by index.
* Efficiency: Memory is used efficiently since elements are stored in contiguous locations.
* Simplicity: Arrays are simple to implement and use, providing straightforward data access and manipulation.

**STEP 2 :**

o Create a class Employee with attributes like employeeId, name, posi5on, and salary

public class Employee {

private String employeeId;

private String name;

private String position;

private double salary;

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

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

public String getEmployeeId() {

return employeeId;

}

public void setEmployeeId(String employeeId) {

this.employeeId = employeeId;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getPosition() {

return position;

}

public void setPosition(String position) {

this.position = position;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

public String toString() {

return "Employee{" +

"employeeId='" + employeeId + '\'' +

", name='" + name + '\'' +

", position='" + position + '\'' +

", salary=" + salary +

'}';

}

}

**STEP 3 :**

*Use an Array to Store Employee Records*

*Implement methods to add, search, traverse, and delete employees in the array.*

import java.util.Arrays;

public class EmployeeManagementSystem {

private Employee[] employees;

private int size;

public EmployeeManagementSystem(int capacity) {

employees = new Employee[capacity];

size = 0;

}

public void addEmployee(Employee employee) {

if (size < employees.length) {

employees[size++] = employee;

} else {

System.out.println("Employee array is full.");

}

}

public Employee searchEmployee(String employeeId) {

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

if (employees[i].getEmployeeId().equals(employeeId)) {

return employees[i];

}

}

return null;

}

public void traverseEmployees() {

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

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

}

}

public void deleteEmployee(String employeeId) {

int index = -1;

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

if (employees[i].getEmployeeId().equals(employeeId)) {

index = i;

break;

}

}

if (index != -1) {

for (int i = index; i < size - 1; i++) {

employees[i] = employees[i + 1];

}

employees[size - 1] = null;

size--;

} else {

System.out.println("Employee with ID " + employeeId + " not found.");

}

}

public static void main(String[] args)

{

EmployeeManagementSystem ems = new EmployeeManagementSystem(20);

ems.addEmployee(new Employee("E101", "David", "Director", 120000));

ems.addEmployee(new Employee("E102", "Emma", "Lead Engineer", 95000));

ems.addEmployee(new Employee("E103", "Fiona", "Consultant", 70000));

System.out.println("List of all employees:");

ems.traverseEmployees();

System.out.println("\nSearching for employee with ID E102:");

Employee foundEmployee = ems.searchEmployee("E102");

System.out.println(foundEmployee);

System.out.println("\nRemoving employee with ID E102:");

ems.deleteEmployee("E102");

System.out.println("\nEmployees list after removal:");

ems.traverseEmployees();

}

}

**STEP 4 :**

**o Analyze the time complexity of each operation (add, search, traverse, delete).**

Time Complexity

Add: O(1) on average, but can be O(n) if resizing is required (not applicable in this simple implementation with a fixed-size array).

Search: O(n) — You need to check each element in the array to find the desired employee.

Traverse: O(n) — You must visit each element in the array.

Delete: O(n) — Finding the element is O(n), and shifting elements to fill the gap is also O(n).

**o Discuss the limitations of array and when to use them.**

Limitations of Arrays

* Fixed Size: Arrays have a fixed size, which can be inefficient if the number of employees changes significantly.
* Insertion/Deletion Complexity: Adding or removing elements involves shifting elements, which can be costly in terms of time complexity.
* Wasted Space: If the array size is overestimated, it can result in wasted memory.

When to Use Arrays

* Small and Fixed-Size Datasets: Arrays are suitable when the number of elements is known and doesn't change frequently.
* Simple Implementation: Arrays are simple and provide constant-time access to elements, making them useful for straightforward tasks with manageable datasets.

**Exercise 5: Task Management System**

**STEP 1 :**

**o Explain the different types of linked lists (Singly Linked List, Doubly Linked List).**

Different Types of Linked Lists

* Singly Linked List:
  + Description: A singly linked list consists of nodes where each node contains data and a reference (or link) to the next node in the sequence. This allows for efficient insertion and deletion at the beginning or middle of the list.
  + Structure: Each node has a data part and a next pointer.
  + Operations:
    - Traversal: Starts from the head and follows the links to visit each node.
    - Insertion/Deletion: Efficient at the beginning or middle, but requires traversal for insertion/deletion at specific positions.
* Doubly Linked List:
  + Description: A doubly linked list extends a singly linked list by adding a reference to the previous node in addition to the next node. This allows traversal in both directions.
  + Structure: Each node has data, next, and prev pointers.
  + Operations:
    - Traversal: Can be done in both forward and backward directions.
    - Insertion/Deletion: More flexible, as you can access both the previous and next nodes.

**STEP 2 :**

**o Create a class Task with attributes like taskId, taskName, and status.**

**public class Task {**

public class Task {

private String taskId;

private String taskName;

private String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

public String getTaskId() {

return taskId;

}

public String getTaskName() {

return taskName;

}

public String getStatus() {

return status;

}

public String toString() {

return "Task{taskId='" + taskId + "', taskName='" + taskName + "', status='" + status + "'}";

}

}

**STEP 3 :**

Singly Linked List Implementation

public class TaskManagementSystem {

private Node head;

private class Node {

Task task;

Node next;

Node(Task task) {

this.task = task;

this.next = null;

}

}

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(String taskId) {

Node current = head;

while (current != null) {

if (current.task.getTaskId().equals(taskId)) {

return current.task;

}

current = current.next;

}

return null;

}

public void traverseTasks() {

Node current = head;

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

public void deleteTask(String taskId) {

if (head == null) {

System.out.println("Task list is empty.");

return;

}

if (head.task.getTaskId().equals(taskId)) {

head = head.next;

return;

}

Node current = head;

while (current.next != null && !current.next.task.getTaskId().equals(taskId)) {

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

} else {

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

}

}

public static void main(String[] args) {

TaskManagementSystem tms = new TaskManagementSystem();

tms.addTask(new Task("T101", "Prepare Presentation", "Pending"));

tms.addTask(new Task("T102", "Review Code", "In Progress"));

tms.addTask(new Task("T103", "Submit Report", "Completed"));

System.out.println("List of all tasks:");

tms.traverseTasks();

System.out.println("\nSearching for task with ID T102:");

Task foundTask = tms.searchTask("T102");

System.out.println(foundTask);

System.out.println("\nRemoving task with ID T102:");

tms.deleteTask("T102");

System.out.println("\nTasks list after removal:");

tms.traverseTasks();

}

}

**STEP 4 :**

**o Analyze the time complexity of each operation.**

Time Complexity

* Add: O(n) — Inserting at the end requires traversing the list if you don’t keep track of the tail node. If you keep a tail reference, it becomes O(1).
* Search: O(n) — You need to traverse the list to find the task.
* Traverse: O(n) — Each element must be visited.
* Delete: O(n) — Requires traversal to find the task, then removal is O(1) if the node is found.

**o Discuss the advantages of linked lists over arrays for dynamic data.**

Advantages of Linked Lists Over Arrays

* Dynamic Size: Linked lists can grow and shrink dynamically, unlike arrays with fixed size. This makes linked lists suitable for situations where the number of elements changes frequently.
* Efficient Insertions/Deletions: Linked lists allow efficient insertions and deletions, especially when operations are performed at the beginning or in the middle of the list. This is because adding or removing nodes only involves updating references.
* No Wasted Space: Linked lists do not require pre-allocation of memory, so there’s no wasted space if the number of tasks is not known in advance.

**Exercise 6: Library Management System**

**STEP 1 :**

**o Explain linear search and binary search algorithms.**

*Linear Search*

Description: Linear search is the simplest search algorithm. It checks each element in a list sequentially until the desired element is found or the end of the list is reached.

Time Complexity:

* Best Case: O(1) — When the desired element is the first in the list.
* Average Case: O(n) — On average, the search might require checking half the elements.
* Worst Case: O(n) — When the desired element is the last one or not present in the list.

Pros:

* Simple and easy to implement.
* Works on unsorted lists.

Cons:

* Inefficient for large lists as it requires checking each element.

*Binary Search*

Description: Binary search is a more efficient algorithm that works on sorted lists. It repeatedly divides the search interval in half and compares the target value to the middle element of the interval.

Time Complexity:

* Best Case: O(1) — When the target element is the middle element.
* Average Case: O(log n) — On average, the search interval halves with each step.
* Worst Case: O(log n) — When the search interval reduces to a single element.

Pros:

* Much faster than linear search for large, sorted lists.
* Reduces the number of comparisons required significantly.

Cons:

* Requires the list to be sorted beforehand.
* More complex to implement compared to linear search.

**STEP 2 :**

**o Create a class Book with attributes like bookId, title, and author**

public class Book {

private String bookId;

private String title;

private String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

public String getBookId() {

return bookId;

}

public String getTitle() {

return title;

}

public String getAuthor() {

return author;

}

public String toString() {

return "Book{bookId='" + bookId + "', title='" + title + "', author='" + author + "'}";

}

}

**STEP 3 :**

Linear Search

public class LinearSearch\_Books {

public static Book linearSearchByTitle(Book[] books, String title)

{

for (Book book : books) {

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

return book;

}

}

return null;

}

}

Binary Search

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch\_Books{

public static Book binarySearchByTitle(Book[] books, String title) {

int left = 0;

int right = books.length - 1;

while (left <= right) {

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

int comparison = books[mid].getTitle().compareToIgnoreCase(title);

if (comparison == 0)

return books[mid];

else if (comparison < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

public static void sortBooksByTitle(Book[] books) {

Arrays.sort(books, Comparator.comparing(Book::getTitle, String.CASE\_INSENSITIVE\_ORDER));

}

}

**STEP 4 :**

**o Compare the time complexity of linear and binary search.**

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)

**o Discuss when to use each algorithm based on the data set size and order.**

When to Use Each Algorithm

* Linear Search:
  + Use Case: When the dataset is small or unsorted. It is simple and works without any preconditions.
  + Pros: Easy to implement and does not require sorting.
  + Cons: Inefficient for large datasets due to linear time complexity.
* Binary Search:
  + Use Case: When the dataset is large and sorted. This is typically more efficient for large datasets because of its logarithmic time complexity.
  + Pros: Much faster than linear search for large datasets if the list is sorted.
  + Cons: Requires the list to be sorted, which adds overhead if sorting is not already done.

**Exercise 7: Financial Forecasting**

**STEP 1 :**

**o Explain the concept of recursion and how it can simplify certain problems.**

Concept of Recursion

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. It is often used to simplify problems that can be divided into similar sub-problems.

Key Components of Recursion:

* Base Case: The condition under which the recursion stops. It prevents infinite recursion.
* Recursive Case: The part of the function where it calls itself with a smaller or simpler problem.

Advantages:

* Simplicity: Makes complex problems easier to understand and implement.
* Reduction in Code: Often results in more concise and readable code.

Disadvantages:

* Stack Overflow: Excessive recursion can lead to stack overflow errors if not properly managed.
* Performance: Recursive solutions can be less efficient due to repeated calculations if not optimized.

**STEP 2 & 3 :**

**o Create a method to calculate the future value using a recursive approach.**

**o Implement a recursive algorithm to predict future values based on past growth rates.**

public class FinancialForecasting {

public static double futureValue(double pv, double r, int n) {

if (n == 0) {

return pv;

}

return futureValue(pv, r, n - 1) \* (1 + r);

}

public static void main(String[] args) {

double presentValue = 2000;

double growthRate = 0.08;

int periods = 12;

double predictedValue = futureValue(presentValue, growthRate, periods);

System.out.println("Predicted future value: " + predictedValue);

}

}

**STEP 4 :**

**o Discuss the time complexity of your recursive algorithm.**

Time Complexity

The time complexity of the recursive `calculateFutureValue` method is (O(n)), where (n) is the number of periods. This is because the function makes (n) recursive calls, each with a reduced number of periods.

Space Complexity: The space complexity is also (O(n)) due to the call stack. Each recursive call adds a new frame to the call stack, proportional to the number of periods.

**o Explain how to optimize the recursive solution to avoid excessive computation.**

Optimization

1. Memoization:

- Concept: Memoization involves storing the results of expensive function calls and reusing these results when the same inputs occur again. This avoids redundant calculations and improves performance.

- Implementation: For this problem, memoization is not typically needed because each calculation is unique and independent. However, in more complex scenarios where subproblems overlap, memoization would be useful.

2. Iterative Approach:

- Concept: Convert the recursive solution into an iterative one to avoid deep recursion and reduce the risk of stack overflow.

- Implementation: An iterative approach uses a loop instead of recursion. It’s generally more efficient for large input sizes.