**SKILL LEARNT: DATA STRUCTURES AND ALGORITHMS (WEEK 1)**

**EXERCISE 2: E-COMMERCE PLATFORM SEARCH FUNCTION**

1. **UNDERSTAND ASYMPTOTIC NOTATION**
2. The Big O Notation where o stands for ‘order of’ is concerned with what happens for every large values of n. It 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 behaviour of a function, not its exact value. It can be used to compare the efficiency of different algorithms or data structures.**

**Big-O Notation helps in analysing the efficiency of algorithms:**

1. It provides a way to describe how the runtime or space requirements of an algorithm grows as the input size increases.
2. It allows programmers to compare different algorithms and choose the most efficient one for a specific problem.
3. Helps in understanding the scalability of algorithms and predicting how they will perform as the input size grows.
4. Enables developers to optimize code and improve overall performance.
5. **BEST CASE RUNNING TIME:** The term ‘best-case performance’ is used to analyse an algorithm under optimal solutions. For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list. However, while developing and choosing an algorithm to solve a problem, we hardly base our decision on the best case performance.

**AVERAGE CASE RUNNING TIME:** The average -case running time of an algorithm is an estimate of the running time for an ‘average’ input. It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution. Average-case running time assumes that all inputs of a given size are equally likely.

**WORST CASE RUNNING TIME**: This denotes the behaviour of an algorithm with respect to the worst possible case of the input instance. The worst- case running time of an algorithm is an upper bound on the running time for any input. Therefore, having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.

**SEARCHING ALGORITHMS AND THEIR TIME COMPLEXITIES**

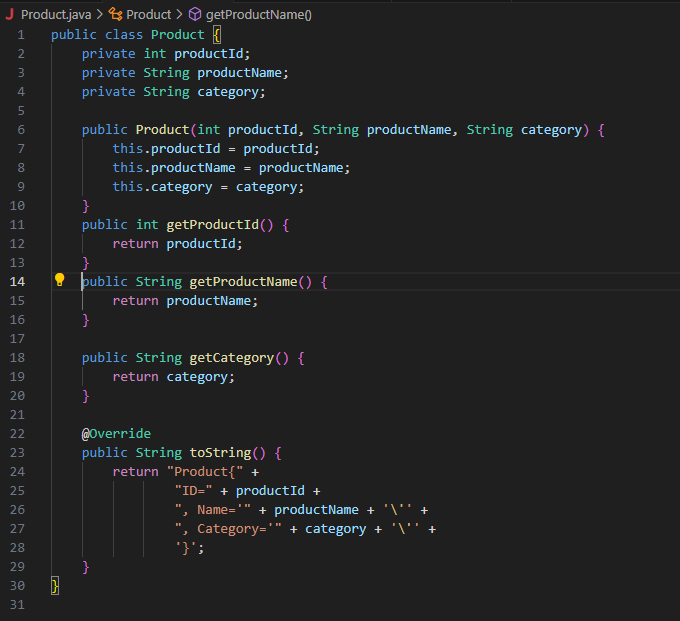
|  |  |  |  |
| --- | --- | --- | --- |
| **ALGORITHMS** | **BEST CASE** | **AVERAGE CASE** | **WORST CASE** |
| 1. **LINEAR SEARCH** | **O(1)** | **O(n)** | **O(n)** |
| 1. **BINARY SEARCH** | **O(1)** | **O(log n)** | **O(log n)** |
| 1. **JUMP SEARCH** | **O(1)** | **O(root n)** | **O(root n)** |
| 1. **INTERPOLATION SEARCH** | **O(1)** | **O(log log n)** | **O(n)** |

1. **SETUP**

We created a Product class with attributes for searching such as productId, productName, and category.

**PRODUCT.JAVA**

The image for the code of Product.java is shown below:



1. **IMPLEMENTATION**

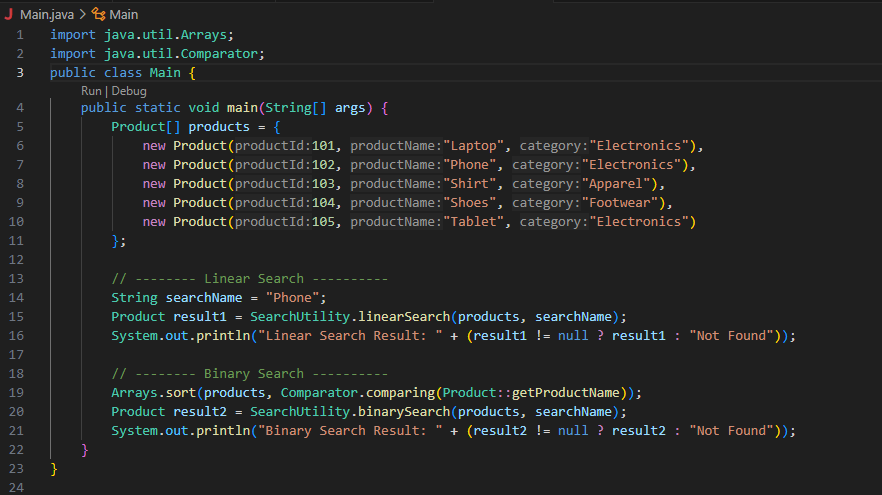
We implemented Linear Search and Binary Search on this:

**SEARCHUTILITY.JAVA**



We also created a Main class for Testing

**MAIN.JAVA**



**OUTPUT:**



1. **ANALYSIS**

|  |  |  |  |
| --- | --- | --- | --- |
| **ALGORITHMS** | **BEST CASE** | **AVERAGE CASE** | **WORST CASE** |
| 1. **LINEAR SEARCH** | O(1) | O(n) | O(n) |
| 1. **BINARY SEARCH** | O(1) | O(log n) | O(log n) |

**WHICH ALGORITHM WORKS BETTER?**

In the context of an **e-commerce platform,** where users frequently search through a large catalog of products, **Binary Search** is generally **more suitable** than Linear Search.

Binary search has several key advantages, especially for large, sorted datasets. First, it is very efficient, with a time complexity of O(log n). This means it significantly cuts down the number of comparisons needed to find an element when compared to linear search. This efficiency makes it perfect for applications that need fast lookups, like e-commerce product searches, database indexing, and real-time systems. Binary search also offers reliable performance, no matter the size of the dataset. It consistently halves the search space with each step. Moreover, it is relatively easy to implement and does not need extra memory, making it both space-efficient and fast. When used with sorted arrays or data structures like balanced trees, binary search provides quick access times, improving the overall performance and responsiveness of an application.

Through this exercise, I learned how to implement and compare two fundamental searching algorithms—**linear search** and **binary search**—within the context of an e-commerce platform. I understood how to design a Java class (Product) with relevant attributes and apply different search strategies on an array of objects. This helped me grasp the practical application of **asymptotic notation** (Big O) to evaluate algorithm efficiency. I also learned the importance of data sorting in binary search and how it significantly improves search performance for large datasets. Overall, this exercise enhanced my problem-solving skills, deepened my understanding of algorithm analysis, and showed me how to select the right algorithm based on real-world requirements such as performance, scalability, and data structure organization.

**EXERCISE 7: FINANCIAL FORECASTING**

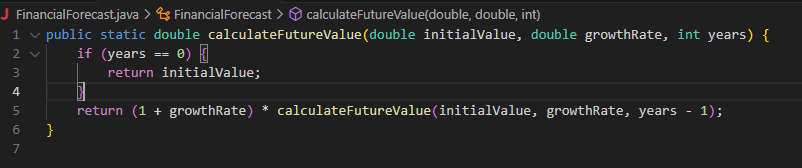
1. **UNDERSTAND RECURSIVE ALGORITHMS**

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. A recursive algorithm makes a step toward a solution and then calls itself to continue the process. It stops when we reach the solution. Because the function can call itself again, this process might go on indefinitely. Therefore, it is important to establish a base case to end the recursion.

Recursion simplifies some problems by breaking them into smaller, manageable sub-problems that follow the same pattern. Instead of using complex loops and handling the stack manually, a recursive function solves a small part of the problem and refers back to itself for the rest. This results in cleaner, more intuitive code, especially for problems like calculating factorials, traversing trees, or generating the Fibonacci sequence. In these cases, the solution builds naturally on smaller versions of the same problem. Recursion removes the complexity of repetitive logic and lets developers focus on defining the base case and the recursive step, making the code easier to understand and maintain.

1. **SETUP**

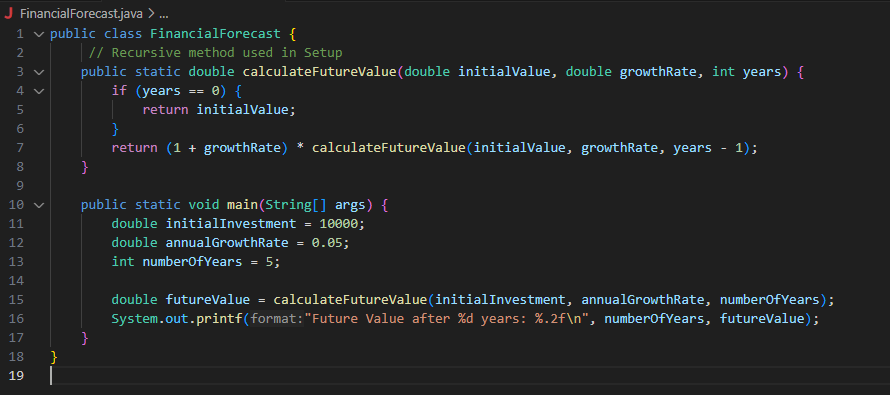
Let’s create a method to calculate the future value using a recursive approach.



1. **IMPLEMENTATION**

We have implemented a recursive algorithm to predict future values based on past growth rates.

**FINANCIALFORECAST.JAVA**



**OUTPUT:**



1. **ANALYSIS**
2. Time complexity

The function calls itself once for each year, reducing the value of years by 1 each time. It continues until it reaches the base case (years == 0). Therefore, the number of recursive calls is exactly equal to the number of years, n.

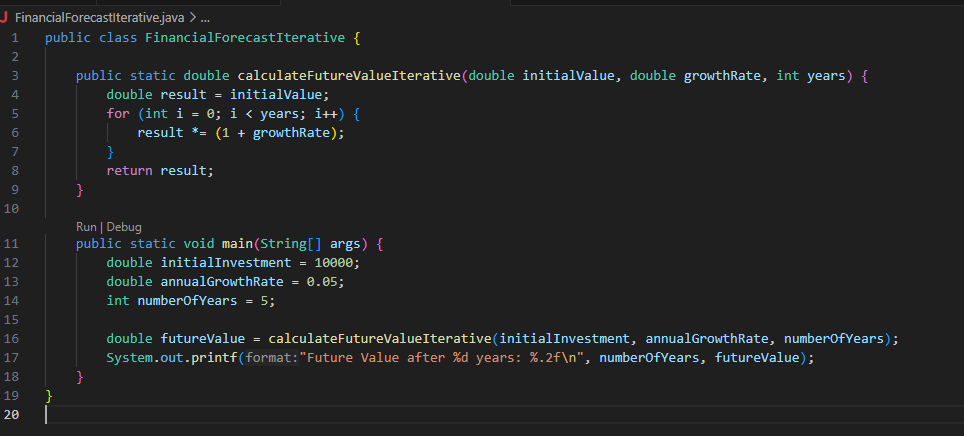
**Time Complexity: O(n),** where **n** is the number of years.

1. Although the recursive solution for future value calculation is simple and elegant, it can lead to **performance issues** such as:
2. **Stack overflow for large input, such as years value.**
3. **Unneeded function call overhead when the same calculation could be done more efficiently.**

**We can use an iterative approach instead.**

Using an iterative approach is better because it avoids stack overflow, uses less memory, and runs faster than recursion. Iteration is easier to understand and debug, especially for simple, linear problems like calculating future value. In Java, recursion isn’t optimized, so using a loop makes the program more efficient and reliable.

**FINANCIALFORECASTITERATIVE.JAVA**



Through this exercise, I learned how recursion can be used to solve problems by breaking them down into smaller sub-problems, and how to implement a recursive method to calculate future financial values based on compound growth. I also understood the limitations of recursion, such as increased memory usage and risk of stack overflow, and how an iterative approach can be a more efficient and reliable alternative for linear problems. Additionally, I gained hands-on experience in analysing time complexity, optimizing recursive solutions, and writing clean, functional Java code for real-world applications like financial forecasting.

**QUIZZES**

1. Can you guess which of the following data structure is linear data structure?

**Answer:** Arrays.

1. The complexity theory does not contain the following case, which is?

**Answer:** Null Case.

1. Which function findings the location of the element with a given value is:

**Answer:** Search.

1. How can you measure the efficiency of the algorithm?

**Answer:** Time and Space.

1. The Average case occurs in the linear search algorithm.

**Answer:** When Item is somewhere in the middle of the array.

1. The complexity of the linear search algorithm is

**Answer:** O(n)

1. Linked lists are the best suited

**Answer:** For the size of the structure and the data in the structure are constantly changing.

1. The complexity of the merge sort algorithm is

**Answer:** O(n log n)

1. How can we measure the time factor when determining the efficiency of the algorithm?

**Answer:** Counting the number of key operations.

1. Arrays are the best data structures

**Answer:** For relatively permanent collections of data.

1. The complexity of the Bubble sort algorithm is

**Answer:** O(n2)

1. Why are the elements of an array are stored successively memory cells?

**Answer:** In this way the computer can keep track only on the address of the first element, and the address of the other elements can be calculated.

1. Can you guess which of the following data structure is NOT a linear data structure?

**Answer:** None of the above.

1. Each array declaration does not give, implicitly or explicitly, the information about which of the following?

**Answer:** The first data from the set that will be stored.

1. The complexity of the Binary search algorithm is

**Answer:** O(log n)

1. When an indirect change of the values of a variable in one module by another module happens, it's called what?

**Answer:** Side effect.