**WEEK 2 : Data Structures and Algorithms**

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

1. **Big O notation** is just a way to understand how **fast** or **slow** your code might be as your data grows.

Big O notation describes the **upper bound** of an algorithm’s runtime as the input size grows. It gives us a high-level understanding of the algorithm’s **efficiency** in terms of **time** and **space**.

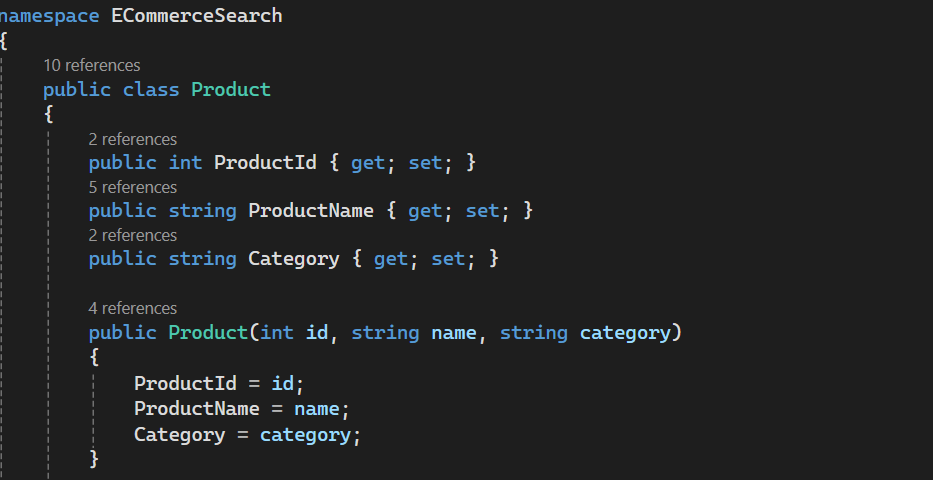
**Example:**

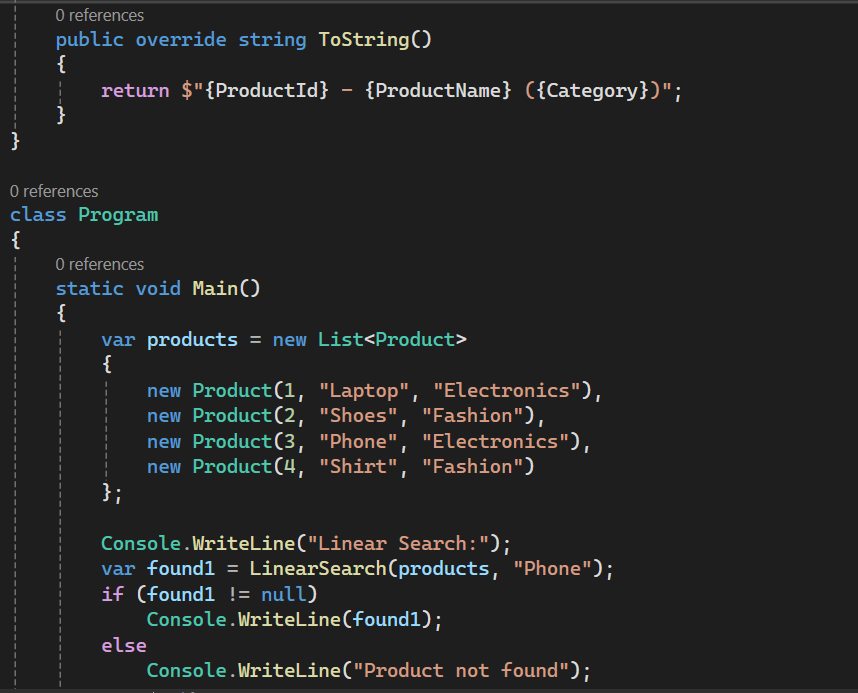
O(1) : Constant time – does not depend on input size.

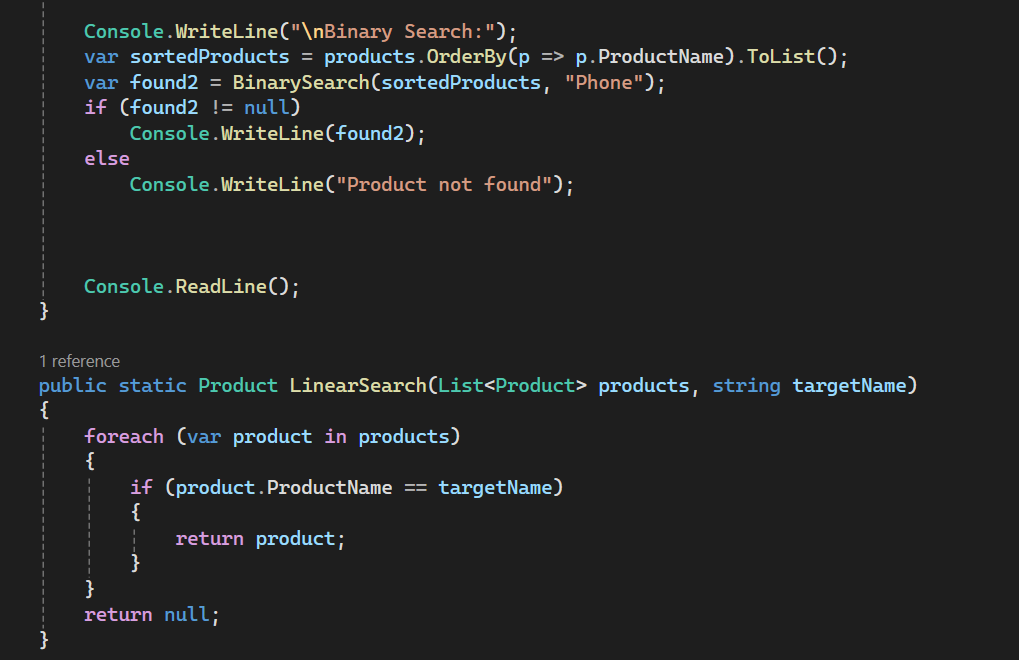
### Best, Average & Worst Case in Searching

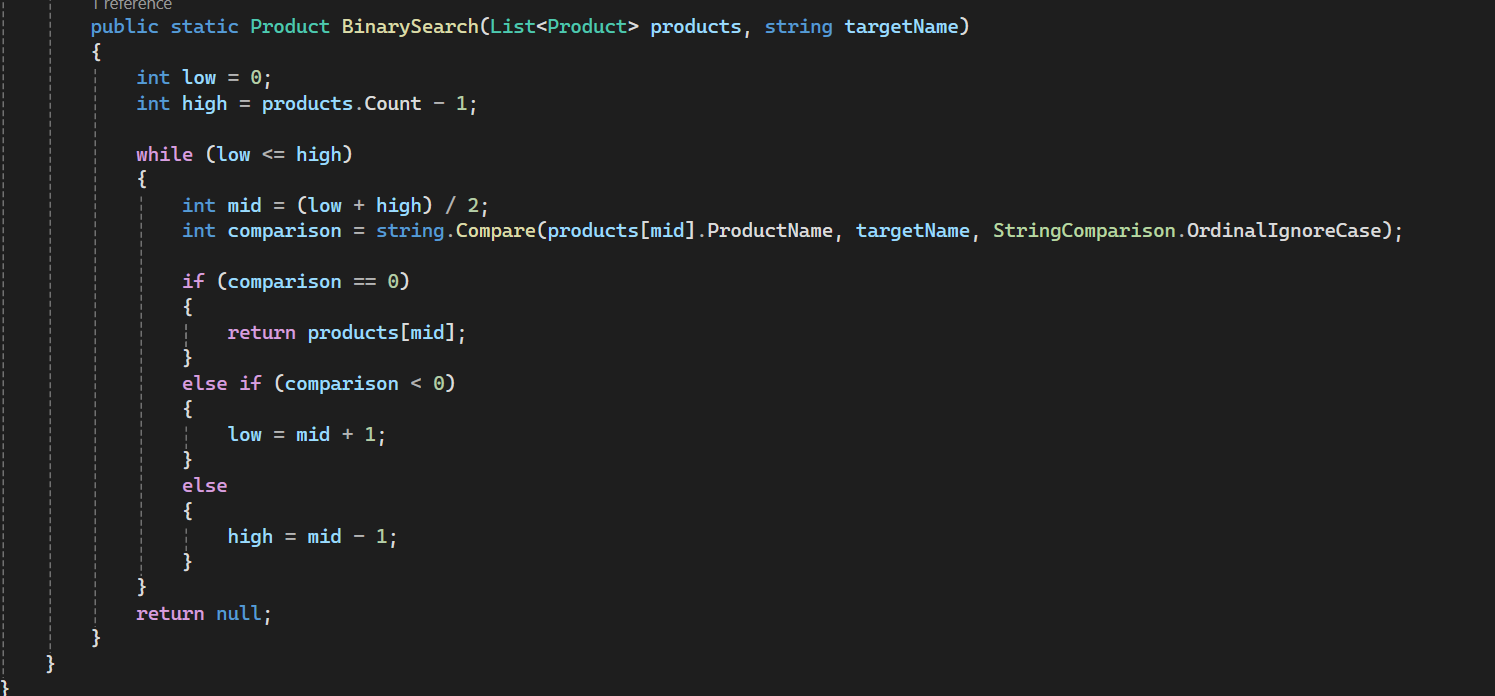
| **Case Type** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Best** | First item → O(1) | Middle item → O(1) |
| **Average** | Middle-ish → O(n/2) | Logarithmic → O(log n) |
| **Worst** | Last or not found → O(n) | Still O(log n) |

1. **& 3.**

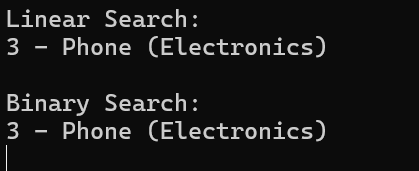








OUTPUT:

****

1. **Analyse**

**Linear Search**: Simple, works on any list, but slow on big data (O(n)). **Linear Search** checks one by one, so it takes more time as the list grows.

**Binary Search**: Super fast (O(log n)), but needs sorted data. **Binary Search** splits the list in half again and again — way faster.

#### For an e-commerce platform where speed and performance matter a lot, **Binary Search is the preferred choice.**

#### **Binary Search is better if:**

* You have **thousands or millions of products**.
* You can keep your product list **sorted** or use a proper **search index**.
* You need **fast search results** for better customer experience.

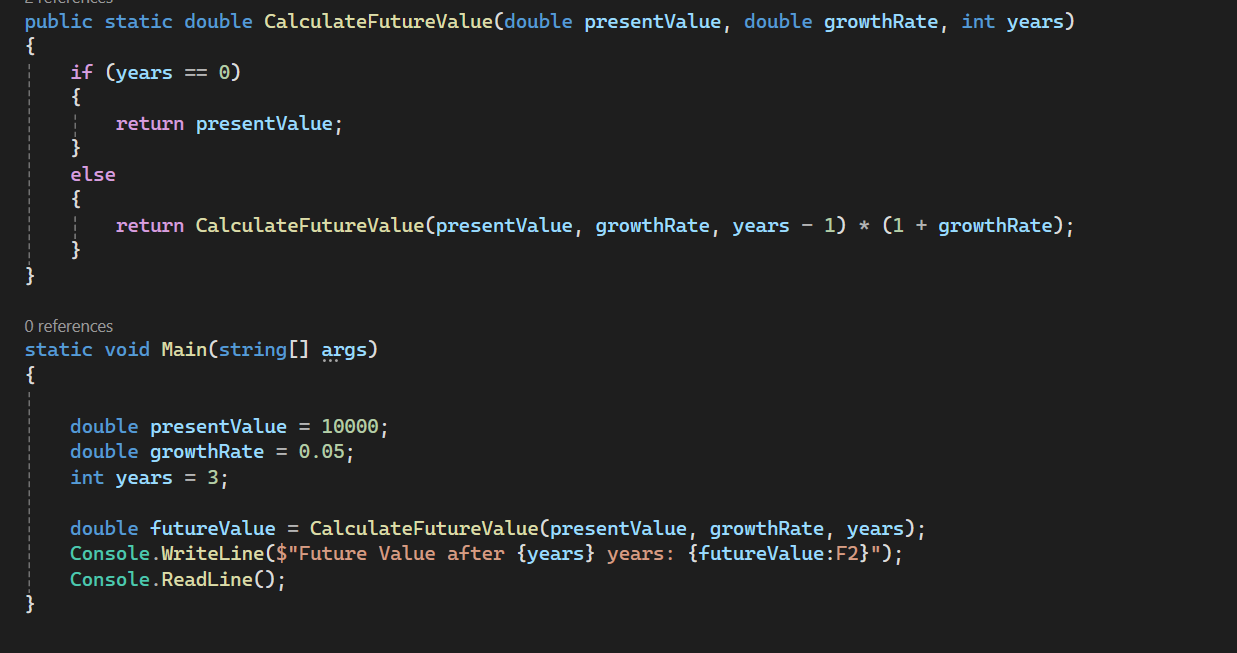
**Linear Search is okay only if:**

* The product list is **very small** (like 10–20 items).
* Or if data is **changing too frequently** and sorting is a hassle

**EXERCISE 7: Financial Forecasting**

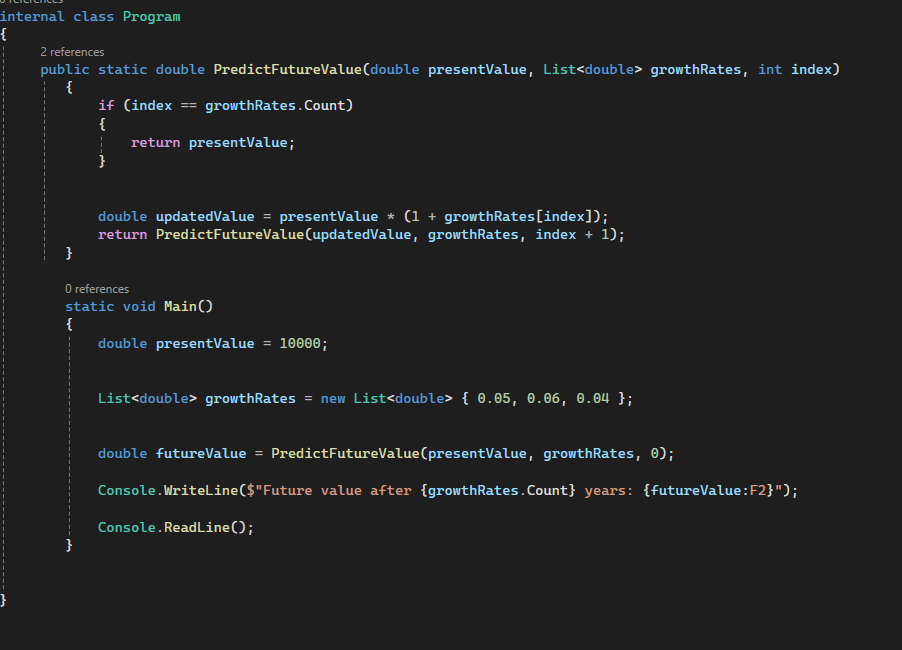
1. Recursion is a programming technique where a function calls itself to solve a problem.

* It simplifies problems by breaking them down into smaller, self-similar subproblems.
* The function continues calling itself with these smaller inputs until a simple base case is reached, at which point the results are combined to solve the original problem.



Output:

Screenshot 2025-06-20 220530



OUTPUT:

Screenshot 2025-06-20 222913

1. **Time Complexity: O(n)**

**Where,**

**n = number of years = growthRates.Count**

The function makes **1 recursive call per year**, so total n calls.

Each call does **basic multiplication**, which is constant time- O(1)

So the total time is **O(n)** — grows linearly with number of years.

Now, to optimize the recursive solution to avoid excessive computation-

* **Use Iteration Instead of Recursion**

Recursion uses more memory (call stack)

Iteration (using a loop) avoids this and is safer for large data.

* **Memoization**

Store previous results to avoid repeating calculations.

Useful in problems like Fibonacci, Not needed here.