Exercise 7: Financial Forecasting

**What is a Recursive Algorithm?**

A recursive algorithm is an algorithm that uses recursion to solve a problem.Recursion is a programming technique where a function calls itself in order to solve smaller instances of the same problem. It’s a way of breaking down complex problems into simpler, more manageable sub-problems.

Recursive algorithms typically have two parts:

1. **Base case:**Which is a condition that stops the recursion.
2. **Recursive case:** Which is a call to the function itself with a smaller version of the problem.

1. Base case:

It is the condition that is written in a recursive function in order for it to get completed and not to run infinitely. After encountering the base condition, the function terminates and returns back to its parent function simultaneously.

Example: In the factorial calculation, the base case is when the input is 0 or 1, for which the factorial is 1.

2. Recursive Case:

The recursive case is the part of a recursive function where the function calls itself with a smaller or simpler input, slowly moving toward the base case.

Example: In factorial calculation, the recursive case is `factorial(n) = n \* factorial(n-1)`.

* factorial(0) = 1 // base case
* factorial(n) = n \* factorial(n - 1) // recursive case

Example: factorial calculation:

public class Factorial {

public static int factorial(int n) {

// Base case

if (n == 0 || n == 1) {

return 1;

}

// Recursive case

return n \* factorial(n - 1);

}

public static void main(String[] args) {

int number = 2;

int result = factorial(number);

System.out.println("Factorial of " + number + " is " + result);

}

}

**How recursion simplifies problem?**

1. Breaking down complex problems: Recursion excels at handling problems that can be divided into smaller, identical instances of the original problem. For example, calculating a factorial or traversing a tree structure can be naturally expressed recursively.

2. Reducing code length: In some cases, recursive solutions can be more concise and easier to understand than iterative (loop-based) solutions, especially when dealing with problems that have a naturally recursive structure.

3. Mirroring problem structure: Recursion can mirror the inherent structure of the problem, making the code easier to map to the problem's logic.

4. Enabling elegance and efficiency: While not always the most efficient approach in terms of performance (especially for very deep recursion), recursion can lead to elegant and readable solutions for certain problems.

**Time complexity of recursive algorithm:**

The time complexity of recursive algorithms depends on two key factors:

* Number of recursive calls made
* Work done in each call

**Optimization to avoid excessive computation:**

**1. Use Iteration Instead of Recursion**

Recursion consumes extra stack memory. You can replace it with a simple loop for better performance.

public static double calculateFutureValueIterative(double principal, double rate, int years) {

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

principal \*= (1 + rate);

}

return principal;

}

Time Complexity: O(n)

Space Complexity: O(1) (more efficient than recursion)

**2. Avoid Unnecessary Memoization**

Memoization stores previously computed values to avoid repetition. However, for simple linear growth like this, where results aren’t reused, memoization adds unnecessary complexity and memory usage. Iteration is more practical here.

**3. Use Direct Mathematical Formula**

The most optimized approach is using the compound interest formula:

public static double calculateFutureValueFormula(double principal, double rate, int years) {

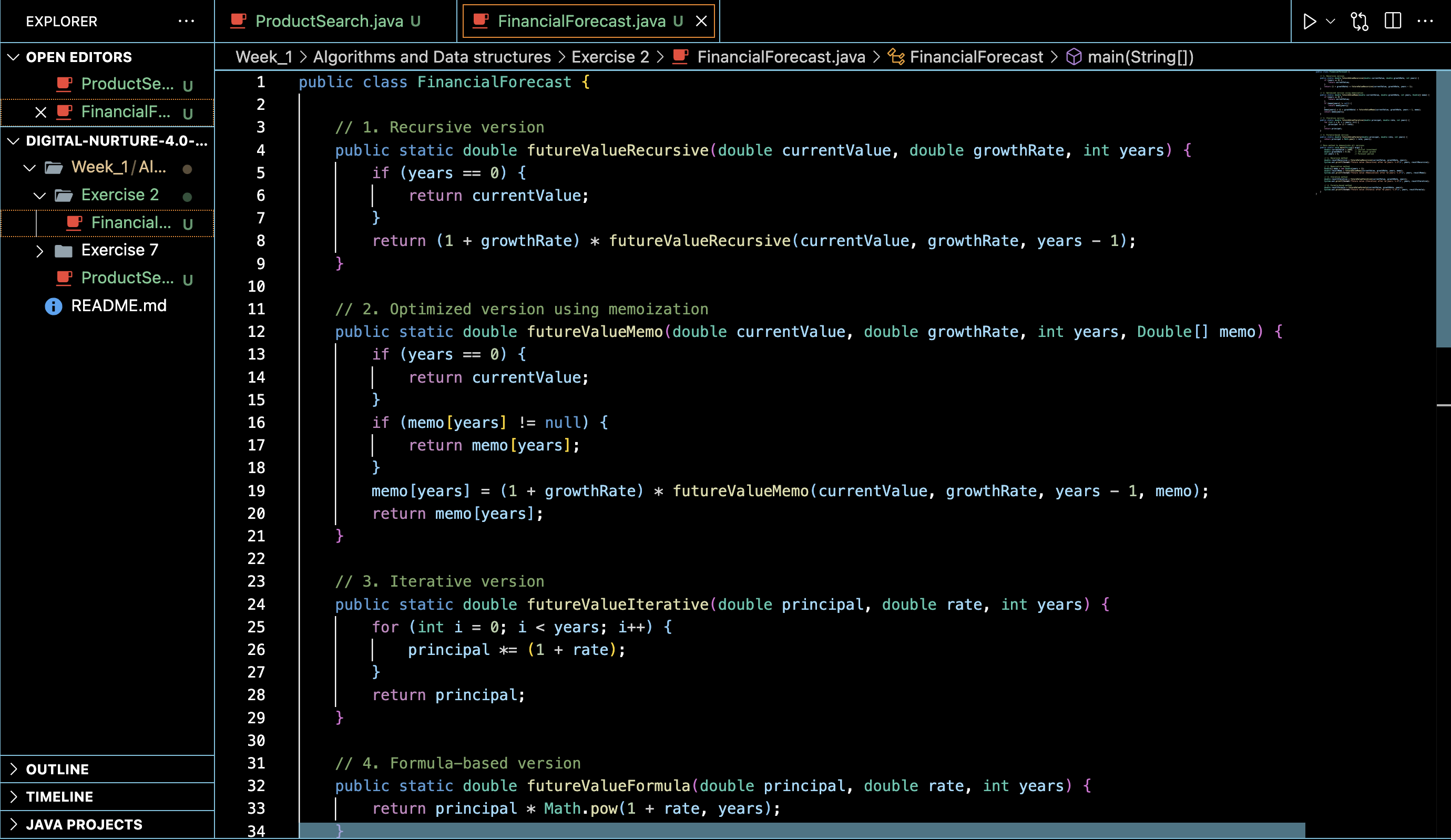
return principal \* Math.pow(1 + rate, years);

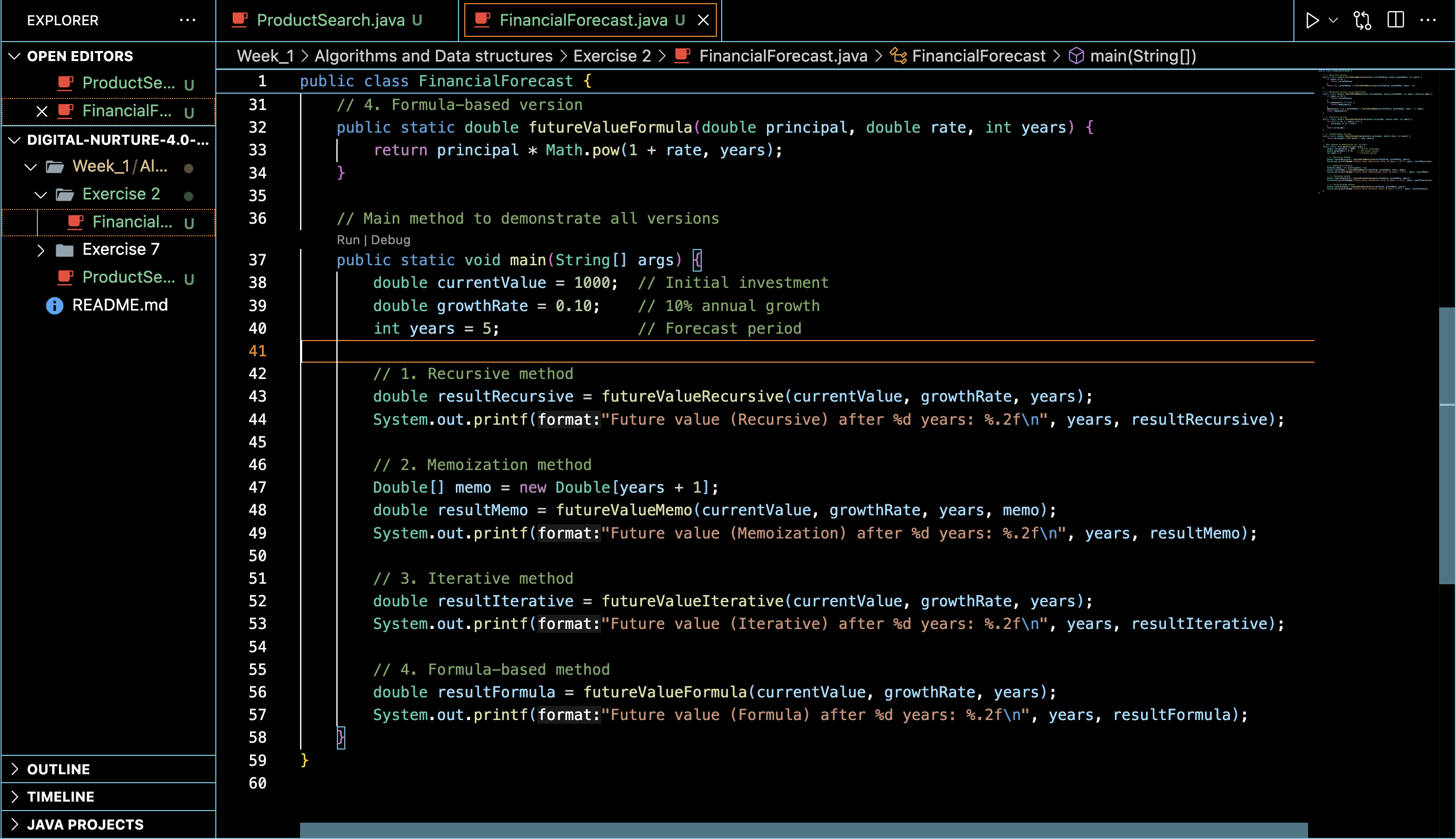
}

Time Complexity: O(1)

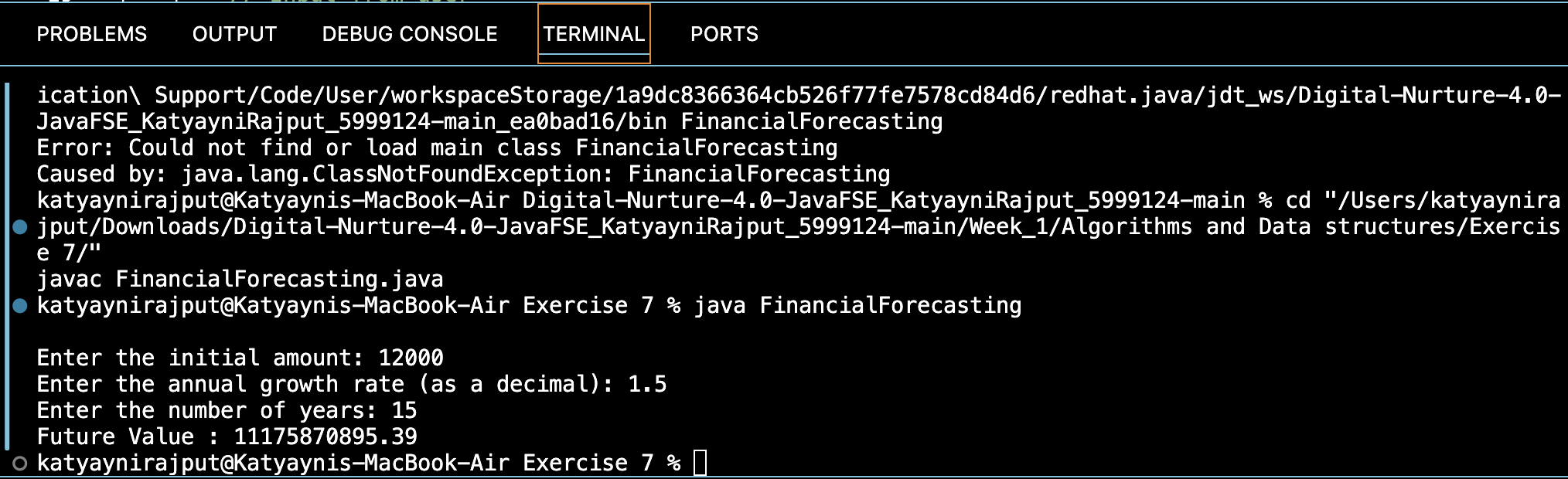
Space Complexity: O(1)

**Setup and Implementation:**

****

****

**Output:**

****