**Recursion** is a programming technique where a function calls itself directly or indirectly to solve a problem.

Instead of solving the entire problem at once, **a recursive function breaks the problem into smaller subproblems** of the same type, and solves them using the same function.

| **Problem Type** | **How Recursion Helps** |
| --- | --- |
| **Divide & Conquer** (e.g. Merge Sort) | Naturally fits breaking down tasks into smaller ones. |
| **Tree Traversals** (e.g. DFS) | Matches the hierarchical structure of trees. |
| **Backtracking** (e.g. Maze solving) | Tries all possible paths with clean rollback (backtrack). |
| **Mathematical Definitions** | Easily translates formulas like Fibonacci or factorial. |

Code:

package DSA2;

public class GrowthPredictor {

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

        if (years == 0) {

            return presentValue;

        }

        double newValue = presentValue \* (1 + rate);

        return calculateFutureValue(newValue, rate, years - 1);

    }

    public static void main(String[] args) {

        double initialValue = 1000.0;

        double growthRate = 0.10;

        int years = 5;

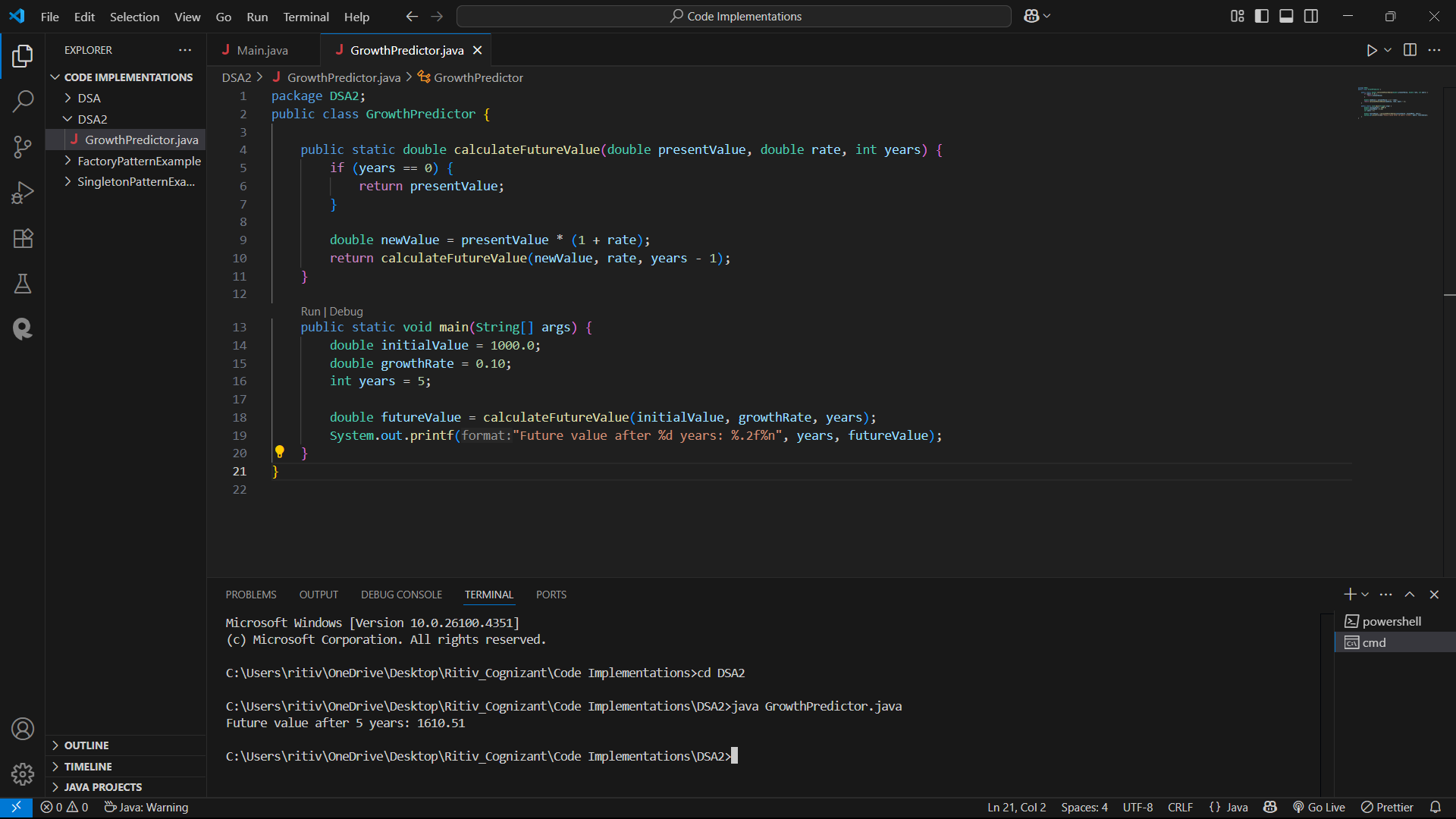
        double futureValue = calculateFutureValue(initialValue, growthRate, years);

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

    }

}

Output:



* **Time Complexity:**  
  The function is called once for each year, so the time complexity is **O(n)**, where n is the number of years.
* **Space Complexity:**  
  Each recursive call adds a new frame to the call stack. So, the space complexity is also **O(n)** due to the recursion depth.

**Time Complexity of the Recursive Algorithm**

The [calculateFutureValue](vscode-file://vscode-app/c:/Users/ritiv/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") method is a simple recursive function. For each year, it makes a recursive call, decrementing [years](vscode-file://vscode-app/c:/Users/ritiv/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) by 1, until [years](vscode-file://vscode-app/c:/Users/ritiv/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) reaches 0.

* **Time Complexity:**  
  The function is called once for each year, so the time complexity is **O(n)**, where n is the number of years.
* **Space Complexity:**  
  Each recursive call adds a new frame to the call stack. So, the space complexity is also **O(n)** due to the recursion depth.

**Optimizing the Recursive Solution**

Although the current recursion is not computationally expensive (no repeated subproblems), recursion can cause stack overflow for large [years](vscode-file://vscode-app/c:/Users/ritiv/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) values. You can optimize by:

1. **Using Iteration Instead of Recursion:**  
   Replace recursion with a loop to avoid stack overhead.
2. **Using the Closed-Form Formula:**  
   The future value can be calculated directly using the formula:  
   [futureValue = presentValue \* Math.pow(1 + rate, years);](vscode-file://vscode-app/c:/Users/ritiv/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)  
   This is **O(1)** time and space.