## **1. What is Recursion, and Why Use It?**

I learned that **recursion** means having a function that calls itself to solve smaller chunks of the same problem. It’s like breaking a big task into little tasks of the same type, until you hit something trivial (a **base case**) and can stop the calls. For example, factorial is a classic:  
 if **n > 0**, compute n \* factorial(n‑1), else return **1**. I realize recursion can make code simpler by matching this "divide‑and‑conquer" idea—but it uses more memory, because every call stays on the stack until it finishes .

## **2. Recursive Future‐Value in Java**

I wrote a recursive method that calculates the future value based on growth rates per year:

class Main {

public static double futureValue(double[] rates, int years, double principal) {

if (years == 0) {

return principal;

}

return futureValue(rates, years - 1, principal) \* (1 + rates[years - 1]);

}

public static void main(String[] args) {

double principal = 1000;

double[] rates = {0.05, 0.04, 0.06, 0.05};

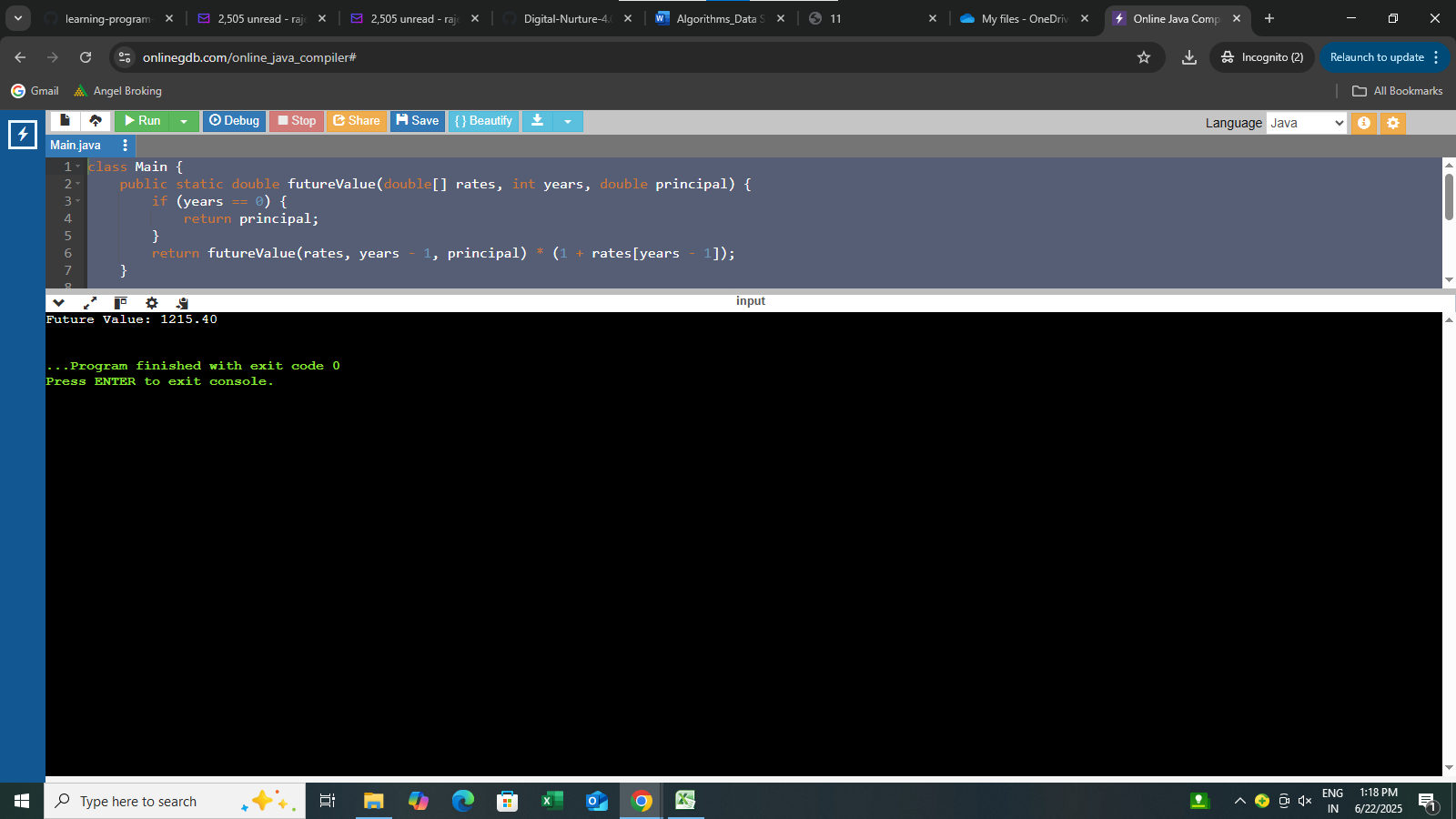
double fv = futureValue(rates, rates.length, principal);

System.out.printf("Future Value: %.2f%n", fv);

}

}

This works by applying each year’s rate one by one until we’ve processed all years.



## **3. How Fast Is This? Time Complexity**

I figured out that this recursion runs in **O(n)** time (where *n* = number of years), because it does one call per year. But space complexity is also **O(n)**, because each function call uses up a stack frame. If *n* is large, this could even cause a **StackOverflowError.**

## **4. How to Make It Better**

### **a) Tail Recursion?**

In tail recursion, the recursive call is the *last* thing the method does. It *can* save stack space—but only *if* the compiler or language supports **tail-call optimization (TCO)**. Sadly, **Java doesn’t do TCO**, so even tail‐recursive methods still use full stack frames and can overflow .

### **b) Use a Loop Instead**

The simplest and safest thing in Java is to rewrite recursion as a loop:

java

CopyEdit

double fv = principal;  
for (double r : rates) {  
 fv \*= (1 + r);  
}

This version still runs in **O(n)** time but uses only **O(1)** space—no risk of stack overflow.

## **5. Final Thoughts**

To summarize:

* Recursion is cool and intuitive for some problems, but it **uses stack memory** and can crash on deep recursions.
* Java doesn’t support optimizing tail recursion—it *always* uses more stack.
* For practical, real‑world tasks in Java (like this forecast), an **iterative version is better**—simpler, safer, and just as fast without the memory risk.