Recursive Algorithm Discussion

How It Works

The algorithm applies compound interest recursively, adjusting the interest rate slightly each year (increasing by 0.001 each year). Here's a breakdown of the helper function:

- **Base Case:** If the current year (cy) equals the target year (y), it returns the final principal amount.
- **Recursive Case:** It calculates the new rate, updates the principal amount, and then calls itself for the next year.

Complexity Analysis

Time Complexity

The time complexity is **O(n)**, where n is the number of years (y). Each recursive call does a constant amount of work, and the function is called n times until the base case is reached.

Space Complexity

Since this is a recursive function, it uses the call stack. Therefore, the space complexity is also **O(n)** due to the recursive depth of n.

Optimizing the Recursive Solution

Use Iteration Instead of Recursion

Recursive solutions can lead to stack overflow for large inputs. An iterative version is more memory efficient:

```
public static double futureValueIterative(double p, double r, int y) {
for (int i = 0; i < y; i++) {
double newRate = r + (0.001 * i);
p *= (1 + newRate);
}
return p;
}
```

- **Time Complexity:** O(n)
- **Space Complexity:** O(1) (no recursion stack)

Memoization

Memoization is helpful for problems with overlapping subproblems (like the Fibonacci sequence). However, in this case, each year depends only on the result of the previous year, with no reuse of intermediate results. Therefore, memoization won't provide significant benefits here.

Conclusion

- Your current recursive function works well for small inputs (e.g., less than 1000 years).
- For better performance and to avoid the risk of stack overflow, prefer the iterative version.
- Memoization is not necessary due to the linear and dependent nature of the computation.