Fibonacci Numbers: A Deep Dive

Stanislav Ostapenko

October 14, 2025

Abstract

This article explores the Fibonacci sequence from basic implementations to advanced mathematical techniques. We begin with a naive recursive method, highlighting its exponential complexity and stack limitations in Java. Using a C++ JVMTI agent, we analyze JVM stack frames to understand StackOverflowException. We address Java's `long` type limitations with BigDecimal for large numbers. Optimization techniques like memoization and dynamic programming are introduced to improve performance. We derive Binet's formula using formal power series and explore matrix exponentiation for logarithmic-time computation. Finally, we discuss real-world applications, including the golden ratio, algorithms, and financial modeling.

Contents

Listings

$$
\begin{aligned}
\mathcal{O}(1) &= \mathcal{O}(\text{yeah}) \\
\mathcal{O}(\log n) &= \mathcal{O}(\text{nice}) \\
\mathcal{O}(n) &= \mathcal{O}(\text{k}) \\
\mathcal{O}(n^2) &= \mathcal{O}(\text{my}) \\
\mathcal{O}(2^n) &= \mathcal{O}(\text{no}) \\
\mathcal{O}(n!) &= \mathcal{O}(\text{mg}) \\
\mathcal{O}(n^n) &= \mathcal{O}(\text{sh*t!})
\end{aligned}
$$

1 Recursion and Mathematical Induction

The Fibonacci sequence, defined as $F_n = F_{n-1} + F_{n-2}$ with $F_0 = 0$ and $F_1 = 1$, is a fundamental concept in mathematics and computer science. Introduced by Leonardo of Pisa in 1202, it appears in nature (e.g., spiral patterns), algorithms (e.g., Fibonacci heaps), and number theory. We'll start our journey from naive recursion to advanced techniques, analyzing their computational complexity and practical limitations.

The concepts of recursion and mathematical induction are closely intertwined, as both rely on solving problems by breaking them down into smaller instances and establishing a base case. Below, we explore their relationship through their structural similarities and shared principles, with a particular emphasis on the role of the base case.

In mathematical induction, the base case establishes the truth of a statement for an initial value. In recursion, the base case is equally critical, as it defines the condition under which the recursive process terminates, returning a specific value without further recursive calls. The base case prevents infinite recursion and provides a foundation for building solutions to larger instances. Without a well-defined base case, a recursive function would continue indefinitely, leading to errors such as stack overflow.

For example, in a recursive factorial function, the base case is typically defined for $n = 0$ or $n = 1$, returning 1. This ensures that the recursion stops at a known value, allowing the algorithm to compute results for larger inputs by building on this foundation.

The base case is the cornerstone of both recursion and mathematical induction:

- **Termination**: In recursion, the base case ensures the process stops, preventing infinite recursion. Without it, the function would attempt to compute values for invalid inputs (e.g., negative numbers) or never terminate.
- **Correctness**: The base case aligns with the mathematical definition of the problem, ensuring accurate results. For factorial, $0! = 1$ and $1! = 1$ are standard definitions.
- **Foundation**: It provides a starting point that recursive calls or inductive steps rely on to build the solution or proof.

Both recursion and mathematical induction rely on the principle of breaking down a problem into simpler components:

- **Mathematical induction** proves a statement for all cases by starting with a base case and using the inductive step to cover all subsequent cases.
- **Recursion** computes a result by solving smaller instances of the same problem, reducing it to the base case.

Recursion and mathematical induction share a fundamental approach: solving or proving something by reducing it to simpler cases, anchored by a well-defined base case. The base case is essential for termination, correctness, and providing a foundation for building solutions or proofs. While induction is a proof technique, recursion is its practical counterpart in programming, with the base case playing a pivotal role in ensuring both processes succeed.

2 Naive Recursion

2.1 Algorithm in Java

The simplest approach to compute Fibonacci numbers is recursion, following the sequence's definition.
Time Complexity: $T(n) = \mathcal{O}(2^n)$
Space Complexity: $M(n) = \mathcal{O}(n)$ (due to call stack depth)

```java
public static int fib0(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib0(n - 1) + fib0(n - 2);
}
```

Listing 1: Naive Recursive Fibonacci in Java

This method is intuitive but inefficient due to redundant calculations, forming a binary recursion tree with approximately 2^n nodes.

2.2 Binet's formula

2.2.1 Intuitive Explanation

Let us think of the Fibonacci sequence not as a list of numbers, but as the sequence of coefficients of a power series. In other words, we define a generating function

$$F(x) = F_0 + F_1 x + F_2 x^2 + F_3 x^3 + \cdots ,$$

where each coefficient corresponds to a Fibonacci number.

Since the Fibonacci sequence satisfies the recurrence relation

$$F_n = F_{n-1} + F_{n-2},$$

we can express this recurrence in terms of $F(x)$ itself. To do that, consider how the series looks when multiplied by x and x^2:

$$\begin{cases} F(x) = F_0 + F_1 x + F_2 x^2 + F_3 x^3 + \cdots , \\ xF(x) = F_0 x + F_1 x^2 + F_2 x^3 + F_3 x^4 + \cdots , \\ x^2 F(x) = F_0 x^2 + F_1 x^3 + F_2 x^4 + F_3 x^5 + \cdots . \end{cases}$$

Now, if we take $F(x) - xF(x) - x^2F(x)$, all the shifted terms cancel out due to the recurrence relation, leaving only the initial conditions:

$$F(x) - xF(x) - x^2 F(x) = F_0 + (F_1 - F_0)x.$$

Assuming $F_0 = 0$ and $F_1 = 1$, we obtain

$$F(x) = \frac{x}{1 - x - x^2}.$$

The denominator here encodes the same recurrence that defines Fibonacci numbers. To understand the structure of $F(x)$, we factor the quadratic polynomial:

$$1 - x - x^2 = (1 - \varphi x)(1 - \psi x),$$

where

$$\varphi = \frac{1 + \sqrt{5}}{2}, \quad \psi = \frac{1 - \sqrt{5}}{2}.$$

By the method of partial fractions, we can decompose $F(x)$ as

$$F(x) = \frac{A}{1 - \varphi x} + \frac{B}{1 - \psi x}.$$

Each term now has a familiar geometric series form:

$$\frac{1}{1 - rx} = 1 + rx + r^2x^2 + r^3x^3 + \cdots ,$$

so we can directly read off the coefficients as powers of φ and ψ.

This is why we deliberately rewrite $F(x)$ in such a form: it allows us to transform an abstract recurrence relation into a closed analytic expression. Eventually, by equating coefficients of x^n, we recover the celebrated Binet formula:

$$F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}.$$

2.2.2 Formal Derivation

We start from the Fibonacci recurrence

$$F_0 = 0, \qquad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2} \quad (n \geq 2).$$

Defining the generating function

$$F(x) = \sum_{n=0}^{\infty} F_n x^n.$$

Then

$$xF(x) = \sum_{n=0}^{\infty} F_n x^{n+1}, \qquad x^2F(x) = \sum_{n=0}^{\infty} F_n x^{n+2}.$$

Applying the recurrence relation

$$F(x) - xF(x) - x^2F(x) = F_0 + (F_1 - F_0)x + \sum_{n=2}^{\infty}(F_n - F_{n-1} - F_{n-2})x^n.$$

Since $F_n - F_{n-1} - F_{n-2} = 0$ for all $n \geq 2$, we get

$$F(x) - xF(x) - x^2F(x) = x.$$

Hence,

$$F(x) = \frac{x}{1 - x - x^2}.$$

Factorization and substitution. Let

$$1 - x - x^2 = (1 - \varphi x)(1 - \psi x),$$

where

$$\varphi = \frac{1 + \sqrt{5}}{2}, \qquad \psi = \frac{1 - \sqrt{5}}{2}.$$

Then

$$F(x) = \frac{x}{(1 - \varphi x)(1 - \psi x)} = A\frac{x}{1 - \varphi x} + B\frac{x}{1 - \psi x}.$$

Solving for constants. Multiplying both sides by $(1 - \varphi x)(1 - \psi x)$:

$$x = A(1 - \psi x) + B(1 - \varphi x) = (A + B) - (\psi A + \varphi B)x.$$

Matching coefficients gives

$$A + B = 0, \qquad \varphi B + \psi A = -1.$$

Solving:

$$A = \frac{1}{\varphi - \psi}, \qquad B = -\frac{1}{\varphi - \psi}.$$

Geometric series expansion

$$\frac{1}{1 - rx} = \sum_{n=0}^{\infty} r^n x^n.$$

Thus,

$$F(x) = \frac{1}{\varphi - \psi}\left(\frac{x}{1 - \varphi x} - \frac{x}{1 - \psi x}\right) = \frac{1}{\varphi - \psi}\sum_{n=1}^{\infty}(\varphi^n - \psi^n)x^n.$$

Extracting coefficients. The coefficient of x^n gives

$$F_n = \frac{\varphi^n - \psi^n}{\varphi - \psi} = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right].$$

$$\boxed{F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}}$$

This is the closed form known as **Binet's formula**.

2.3 Time Complexity (Big $\mathcal{O}$)

The recursive algorithm generates a binary recursion tree, where each node for $n \geq 2$ spawns two child nodes: $F(n-1)$ and $F(n-2)$. The total number of function calls corresponds to the number of nodes in the recursion tree. For a given n, the tree has a depth of approximately n, and the number of nodes grows exponentially. The recurrence relation for the number of operations $T(n)$ is:

$$T(n) = T(n-1) + T(n-2) + O(1),$$

where $O(1)$ accounts for the constant-time addition operation. The base cases are:

$$T(0) = O(1), \quad T(1) = O(1).$$

This recurrence is similar to the Fibonacci sequence itself. The number of nodes is approximately $2F(n) - 1$, where $F(n) \approx \phi^n/\sqrt{5}$, and $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio. Thus, the time complexity is:

$$T(n) = O(\phi^n) \approx O(1.618^n).$$

2.4 Empirical Validation of Time Complexity

Let's calculate execution time of first 50 Fibonacci numbers. Also, save exec results in CSV file further for analysis.

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.concurrent.TimeUnit;

public class RecursiveGrowthDemonstrator {
    public static long fibonacciRecursive(int n) {
        if (n <= 1) {
            return n;
        }
        return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
    }

```

```
    public static void main(String[] args) {
        String filename = "fibonacci_data.csv";
        int last_n = 50;
        try (PrintWriter writer = new PrintWriter(new FileWriter(filename))) {
            // Write the CSV file header
            writer.println("n,Fn,time_sec");
            for (int n = 1; n <= last_n; n++) {
                long startTime = System.nanoTime();
                long result = fibonacciRecursive(n);
                long endTime = System.nanoTime();

                double durationSec = (double) (endTime - startTime) / 1_000_000_000.0;
                writer.printf("%d,%d,%.10f%n", n, result, durationSec);
                System.out.printf("F(%d) calculated in %.4f sec.%n", n, durationSec);
            }
        } catch (IOException e) {
            System.err.println("Error while writing to file: " + e.getMessage());
        }
    }
}
```

To show how the time grows, let's build a chart -

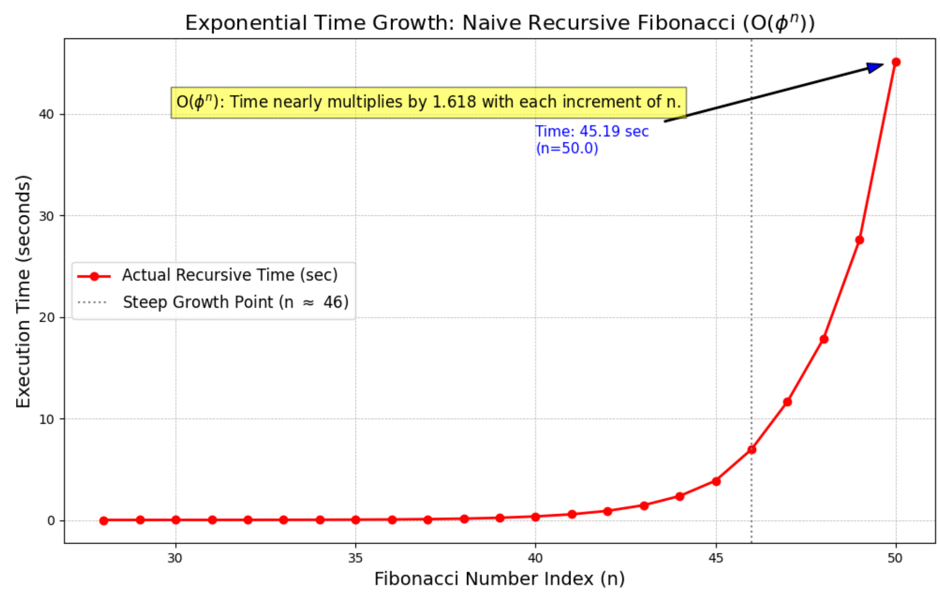

Figure 1: Exponent execution time

To produce this image we use Python with some Pandas :

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import os

# --- 1. Define the filename and check for its existence ---
FILENAME = 'fibonacci_data.csv'

```

```
if not os.path.exists(FILENAME):
    print(f"Error: File '{FILENAME}' not found.")
    print("Please run the Java code first to generate the data file.")
    exit()

# --- 2. Read and Prepare Data ---
try:
    # Read the data from the CSV file
    df = pd.read_csv(FILENAME)
except Exception as e:
    print(f"Error reading CSV file: {e}")
    exit()

# Filter out very small execution times (mostly for n < 30)
# as they introduce noise, focusing the graph on the exponential growth phase.
# We'll keep only data points where time is greater than 1 millisecond (0.001 sec).
df_filtered = df[df['time_sec'] > 0.001].copy()

if df_filtered.empty:
    print("Not enough data points with significant execution time (above 0.001 sec) to plot exponential growth.")
    print("Try increasing the 'last_n' value in your Java code (e.g., to 45).")
    exit()


# --- 3. Plotting the Exponential Growth ---
plt.figure(figsize=(12, 7))

# Plot the actual recursive time
plt.plot(df_filtered['n'], df_filtered['time_sec'],
         marker='o', linestyle='-', color='red', label='Actual Recursive Time (sec)', linewidth=2)

plt.title('Exponential Time Growth: Naive Recursive Fibonacci (O($\phi^n$))', fontsize=16)
plt.xlabel('Fibonacci Number Index (n)', fontsize=14)
plt.ylabel('Execution Time (seconds)', fontsize=14)
plt.legend(fontsize=12)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)

# Highlight the steep rise for visual emphasis
if len(df_filtered) > 5:
    steep_start_n = df_filtered[df_filtered['time_sec'] > df_filtered['time_sec'].max() * 0.1]['n'].min()
    plt.axvline(x=steep_start_n, color='gray', linestyle=':', linewidth=1.5,
                label=f'Steep Growth Point (n $\\approx$ {steep_start_n})')
    plt.legend(fontsize=12)


# --- 4. Adding Annotations for Educational Value ---

# Find the last calculated point
last_n_point = df_filtered.iloc[-1]
plt.annotate(
    f'Time: {last_n_point["time_sec"]:.2f} sec\n(n={last_n_point["n"]})',
    xy=(last_n_point['n'], last_n_point['time_sec']),
    xytext=(last_n_point['n'] - 10, last_n_point['time_sec'] * 0.8),
    arrowprops=dict(facecolor='blue', shrink=0.05, width=1, headwidth=8),
    fontsize=11,
```

```python
    color='blue'
)

# Text explanation on the graph
plt.text(df_filtered['n'].min() + 2, df_filtered['time_sec'].max() * 0.9,
         'O($\phi^n$): Time nearly multiplies by 1.618 with each increment of n.',
         fontsize=12, bbox=dict(facecolor='yellow', alpha=0.5))

plt.savefig("exponential-time-growth.png")
plt.show()
```

Now we are interested in exact formula of this type of growth. To achieve our goal we'll going to use SciPy.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Define the exponential model function
def exponential_model(n, a, b):
    return a * b**n

# Read data from CSV file
data = pd.read_csv('fibonacci_data.csv')

# Extract n and time_sec columns
n = data['n'].values
time_sec = data['time_sec'].values

# Fit the exponential model
popt, pcov = curve_fit(exponential_model, n, time_sec, p0=[1e-6, 1.618])  # Initial guess: a=1e-6, b=1.618
a, b = popt
print(f"Fitted model: T(n) = {a:.10f} * {b:.6f}^n")

# Compute predicted time values
predicted_time = exponential_model(n, a, b)

# Plot the results
plt.figure(figsize=(10, 6))
plt.scatter(n, time_sec, color='blue', label='Experimental data')
plt.plot(n, predicted_time, color='red', label=f'Model: T(n) = {a:.2e} * {b:.6f}^n')
plt.xlabel('n')
plt.ylabel('Execution time (sec)')
plt.title('Execution time of recursive Fibonacci algorithm')
plt.yscale('log')  # Log scale for better visualization of exponential growth
plt.legend()
plt.grid(True)
plt.savefig("experimental-data-formula.png")
plt.show()

# Compare b to the golden ratio
phi = (1 + np.sqrt(5)) / 2
print(f"Golden ratio φ(): {phi:.6f}")
print(f"Deviation of b from φ: {abs(b - phi):.6f}")
```

Which shows us the following :

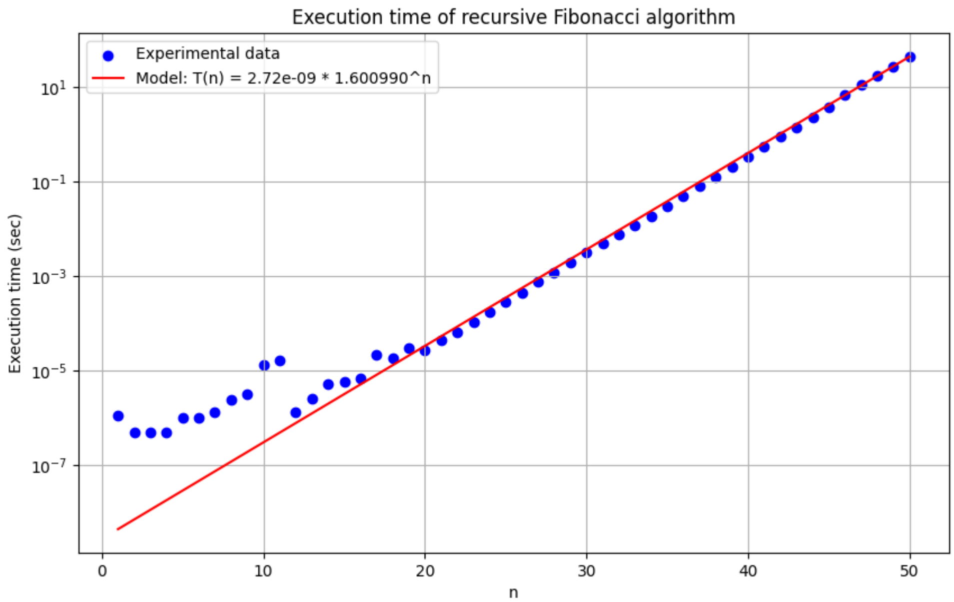

Figure 2: Exponent execution time

Axes made logarithmic for better looking chart. So, we have result of out numerical modeling

```
python empiristic-formula-for-time-complexity.py
Fitted model: T(n) = 0.0000000027 * 1.600990^n
Golden ratio φ(): 1.618034
Deviation of b from φ: 0.017044
```

So, we can have exec time needed to calculate n-th fib. It grows fast and could be very big.

Table 1: Estimated execution time of recursive Fibonacci algorithm using the model $T(n) = 2.7 \times 10^{-9} \cdot 1.600990^n$

n	$T(n)$	Unit
20	3.30e-05	sec
30	3.66e-03	sec
40	4.04e-01	sec
50	4.48e+01	sec
60	1.3752	hr
70	6.3395	days
80	1.9215	yr
90	212.5851	yr
100	23519.0125	yr

So, we have kind of bad news here. To calculate $F(100)$ we'll need ≈ 25000 years. Too long to wait, soon we improve algorithm.

2.5 Introduction to JVM Memory Structures

The Java Virtual Machine (JVM) is a runtime environment that executes Java bytecode, enabling platform-independent execution of Java programs. The JVM manages memory through a structured architecture that supports dynamic allocation, thread execution, and garbage collection.

The JVM divides its memory into several regions, each serving a specific purpose in program execution. These regions are broadly categorized into **per-thread** and **shared** areas:

- **Per-Thread Areas**: Allocated for each thread to ensure isolation and manage method execution.
- **Shared Areas**: Accessible by all threads for storing objects and class metadata.

Memory management is critical for performance, as it affects allocation speed, garbage collection, and thread synchronization. The JVM's memory model is defined by the Java Virtual Machine Specification (JVMS).

The JVM's memory is organized into the following key areas:

1. **Heap**:
   - A shared memory region where all objects and arrays are allocated using the `new` keyword.
   - Divided into:
     - **Young Generation** (Eden and Survivor spaces): For newly created objects, managed by frequent minor garbage collections.
     - **Old Generation**: For long-lived objects, managed by less frequent major garbage collections.
     - **Metaspace** (Java 8+): Stores class metadata, replacing the Permanent Generation (pre-Java 8).
   - Configurable via flags like `-Xmx` (maximum heap size) and `-Xms` (initial heap size).
2. **Java Stack**:
   - A per-thread memory area that stores **call frames** for method invocations.
   - Each frame contains a local variable array, operand stack, and frame data (e.g., program counter, return address).
   - Size is configurable via `-Xss`. Excessive recursion can cause a `StackOverflowError`.
3. **Program Counter (PC) Register**:
   - A per-thread register that holds the address of the current bytecode instruction being executed.
   - Points to the current instruction in the active call frame's bytecode.
4. **Method Area**:
   - A shared area that stores class metadata, including bytecode, constant pools, and method tables.
   - In Java 8+, the Method Area is implemented as Metaspace, which uses native memory rather than the heap.
5. **Native Method Stack**:
   - A per-thread stack for executing native methods (e.g., C/C++ code called via JNI).
   - Similar to the Java stack but tailored for non-Java code.

Consider this Java code:

```java
public class Example {
 public static void main(String[] args) {
  String str = new String("Hello");
  int result = add(3, 4);
  System.out.println(str + result);
 }
 public static int add(int a, int b) {
  return a + b;
 }
}
```

Memory Region	Type	Purpose
Heap	Shared	Stores objects, arrays, and class metadata (Metaspace in Java 8+)
Java Stack	Per-thread	Stores call frames for method execution
PC Register	Per-thread	Tracks current bytecode instruction
Method Area	Shared	Stores class metadata and constant pools
Native Method Stack	Per-thread	Manages native method execution

Table 2: JVM Memory Regions

1. **Heap**: The `String` object `"Hello"` is allocated in the heap's Eden space.
2. **Java Stack**: The `main` method's call frame stores the `str` reference and `args`. A new frame for `add` stores parameters `a` and `b`.
3. **PC Register**: Tracks the current bytecode instruction in `main` or `add`.
4. **Method Area**: Stores the bytecode and constant pool for `Example` class, including the `"Hello"` string literal.
5. **Garbage Collection**: After `main` ends, the `String` object may be reclaimed if no references remain.

The memory structures work together to support JVM execution:

- **Thread Isolation**: Per-thread areas (Java Stack, PC Register, Native Method Stack) ensure threads execute independently without interference.
- **Shared Resources**: The heap and Method Area (or Metaspace) allow threads to share objects and class data, requiring synchronization (e.g., `synchronized` blocks) to avoid race conditions.
- **Garbage Collection**: The garbage collector scans the heap, using references from stacks, Method Area, and static fields as roots to identify reachable objects.

Key Concepts :

- **Heap vs. Stack**: Heap stores dynamic, shared objects; stacks store method-scoped, thread-specific data.
- **Garbage Collection**: Automatically reclaims heap memory but requires careful reference management to avoid leaks.
- **Performance**: Memory size tuning (e.g., `-Xmx`, `-Xss`) impacts performance. Large heaps or stacks may slow execution or garbage collection.
- **Debugging**: Tools like `jstack` (for stacks), `jmap` (for heap), and VisualVM help analyze memory usage and diagnose issues.

2.6 Method Execution in Java

2.6.1 Core concepts

Java's method invocation and parameter passing mechanisms are central to understanding its runtime behavior in the Java Virtual Machine (JVM). Java exclusively uses **pass-by-value** for all parameter passing, impacting both iterative and recursive methods. Recursion, including **tail recursion**, interacts with the

JVM's stack and heap, while Java's lack of **tail call optimization (TCO)** affects performance in deep recursion.

- **Pass-by-Value**: The method receives a copy of the argument's value (primitive or object reference). Changes to the parameter do not affect the caller's variable.
- **Pass-by-Reference**: The method receives a reference to the original argument's memory location, so changes directly modify the caller's variable.

Java uses **pass-by-value** exclusively. For **primitive types** (e.g., `int`, `double`), the value is copied. For **object references**, the reference (not the object) is copied, allowing modification of the object's state in the heap but not reassignment of the caller's reference.

2.6.2 Pass-by-Value in Java

When a method is called, the JVM creates a call frame on the thread's Java stack, copying arguments into the frame's local variable array:

- **Primitive Types**: The value (e.g., 5 for an `int`) is copied. Modifying the parameter changes only the local copy.
- **Object References**: The reference (memory address to a heap object) is copied. Modifying the object's state (e.g., fields) affects the heap, visible to all references. Reassigning the reference (e.g., `obj = new Object()`) is local.

This behavior applies to both iterative and recursive methods, but recursion increases stack depth, risking `StackOverflowError` for deep calls.

Consider a Java program demonstrating pass-by-value in both iterative and recursive contexts:

```java
public class Example {
 public static void main(String[] args) {
  int num = 5;
  StringBuilder sb = new StringBuilder("Factorial: ");
  modifyPrimitive(num);
  modifyObject(sb);
  int result = factorialTail(num, 1, sb);
  System.out.println("num: " + num); // Outputs: num: 5
  System.out.println("sb: " + sb + " " + result); // Outputs: sb: Factorial: 5*4*3*2*1 120
 }

 public static void modifyPrimitive(int x) {
  x = 10; // Modifies local copy
 }

 public static void modifyObject(StringBuilder builder) {
  builder.append("World"); // Modifies heap object
  builder = new StringBuilder("New"); // Local reassignment
 }

 public static int factorialTail(int n, int acc, StringBuilder log) {
  if (n <= 1) {
   log.append("1");
   return acc;
  }
  log.append(n + "*");
  return factorialTail(n - 1, n * acc, log); // Tail-recursive call
 }
}
```

1. **Primitive (`num`)**: In `modifyPrimitive`, `x` is a copy of `num` (5). Setting `x = 10` affects only the local copy, so `num` remains 5.
2. **Object Reference (`sb`)**: In `modifyObject`, `builder` is a copy of the reference to `StringBuilder`. `builder.append("World")` modifies the heap object, affecting `sb`. Reassigning `builder = new StringBuilder("New")` is local, so `sb` retains its reference.
3. **Tail Recursion (`factorialTail`)**: Each recursive call copies `n`, `acc`, and `log`. Modifications to `log` (e.g., `log.append(n + "*")`) persist in the heap. The recursive call is the last operation, but Java creates a new frame each time, risking stack overflow for large `n`.

2.6.3 Tail Recursion

A method is **tail-recursive** if the recursive call is the final operation, with no pending computations. In languages with **tail call optimization (TCO)**, the runtime reuses the current frame, avoiding stack growth. Java's JVM does not support TCO, so each recursive call creates a new frame, copying arguments via pass-by-value. It's because JVM prioritizes general-purpose execution and accurate stack traces for debugging over TCO.
TCO support varies across languages, impacting recursion efficiency:

- **Java**: No TCO; each call adds a frame, risking `StackOverflowError`.
- **Scala**: TCO for self-recursive calls with `@tailrec`, compiling to loops on the JVM.
- **Python**: No TCO; uses iteration or trampolining for deep recursion.
- **JavaScript**: Partial TCO (e.g., Safari supports it, V8 does not).
- **Haskell**: Full TCO with lazy evaluation, ideal for recursion-heavy code.

Language	TCO Support	Workaround
Java	None	Iteration
Scala	Yes (`@tailrec`)	None needed
Python	None	Iteration
JavaScript	Partial	Iteration
Haskell	Full	None needed

Table 3: Tail Call Optimization Across Languages

2.7 Depth and `StackOverflow`

Recursive calls create stack frames in the JVM, which can lead to a `StackOverflowError`. We demonstrate this with a simple recursive program :

```
public class RecursionDepth {
 private static int depth = 0;

 public static void recurse() {
  depth++;
  recurse();
 }

 public static void main(String[] args) {
  try {
   recurse();
  } catch (StackOverflowError e) {
   System.out.println("Max recursion depth: " + depth);
```

```
    }
  }
}
```

Listing 2: Testing Recursion Depth in Java

Run with:

```
java -Xss1m RecursionDepth
```

Listing 3: Running RecursionDepth

It means that even without calculating something, we limited by the value of stack. God news is that it could be increased, but we have no clue how much we need.

2.8 Recursion Tree

```
public class Simple {
 static int depth = 0;

 public static int fib0(int n) throws Exception{
  depth++;
  System.out.println("fib0(" + n + ") depth=" + depth + " frames=" +
      Thread.currentThread().getStackTrace().length);

  if (n == 0) {
   depth--;
   return 0;
  }
  if (n == 1) {
   depth--;
   return 1;
  }

  int result = fib0(n - 1) + fib0(n - 2);
  depth--;
  return result;
 }

 public static void main(String[] args) throws Exception {
  System.out.println("Result: " + fib0(5));
 }
}
```

We have to have compare depth calculation and number of frames, make conclusions.

Output :

```
C:\temp\fibonacci-article>java Simple
fib0(5) depth=1 frames=3
fib0(4) depth=2 frames=4
fib0(3) depth=3 frames=5
fib0(2) depth=4 frames=6
fib0(1) depth=5 frames=7
fib0(0) depth=5 frames=7
fib0(1) depth=4 frames=6
fib0(2) depth=3 frames=5
fib0(1) depth=4 frames=6
fib0(0) depth=4 frames=6
fib0(3) depth=2 frames=4
fib0(2) depth=3 frames=5
```

```
fib0(1) depth=4 frames=6
fib0(0) depth=4 frames=6
fib0(1) depth=3 frames=5
Result: 5
```

Let's make an improvement. We'll all ident to previous code according to depth

```
public class SimpleIdent {
 static int depth = 0;

 public static int fib0(int n) throws Exception {
  // print with indentation
  String indent = "  ".repeat(depth);
  System.out.println(indent + "fib0(" + n + ") depth=" + depth);

  depth++;
  int result;
  if (n == 0) result = 0;
  else if (n == 1) result = 1;
  else result = fib0(n - 1) + fib0(n - 2);
  depth--;

  System.out.println(indent + "=> fib0(" + n + ") = " + result);
  return result;
 }

 public static void main(String[] args) throws Exception {
  System.out.println("Result: " + fib0(5));
 }
}
```

Output is better. BTW code is good for debugging any recursion.

```
fib0(5) depth=0
  fib0(4) depth=1
    fib0(3) depth=2
      fib0(2) depth=3
        fib0(1) depth=4
        => fib0(1) = 1
        fib0(0) depth=4
        => fib0(0) = 0
      => fib0(2) = 1
      fib0(1) depth=3
      => fib0(1) = 1
    => fib0(3) = 2
    fib0(2) depth=2
      fib0(1) depth=3
      => fib0(1) = 1
      fib0(0) depth=3
      => fib0(0) = 0
    => fib0(2) = 1
  => fib0(4) = 3
  fib0(3) depth=1
    fib0(2) depth=2
      fib0(1) depth=3
      => fib0(1) = 1
      fib0(0) depth=3
      => fib0(0) = 0
    => fib0(2) = 1
    fib0(1) depth=2
    => fib0(1) = 1
  => fib0(3) = 2
=> fib0(5) = 5
Result: 5
```

We can do better. Let's buid a tree using dot syntax (blah-blah-blah)

```
import java.io.FileWriter;
import java.io.IOException;

public class Simple3 {

 static class NodeId {
  int id;
  NodeId(int id) { this.id = id; }
 }

 static int idCounter = 0;

 public static int fib0(int n, FileWriter fw, NodeId parent) throws IOException {
  int myId = idCounter++;
  fw.write(String.format("  node%d [label=\"fib(%d)\"];\n", myId, n));

  if (parent != null) {
   fw.write(String.format("  node%d -> node%d;\n", parent.id, myId));
  }

  int result;
  if (n == 0) result = 0;
  else if (n == 1) result = 1;
  else {
```

```java
            int left = fib0(n - 1, fw, new NodeId(myId));
            int right = fib0(n - 2, fw, new NodeId(myId));
            result = left + right;
        }

        return result;
    }

    public static void main(String[] args) throws IOException {
        FileWriter fw = new FileWriter("fib_tree.dot");
        fw.write("digraph G {\n");
            fib0(5, fw, null);
            fw.write("}\n");
        fw.close();
        System.out.println("DOT file generated: fib_tree.dot");
    }
}
```

Convert it to PNG :

```
C:\temp\>dot -Tpng fib_tree.dot -o fib_tree.png
```

And here it is :

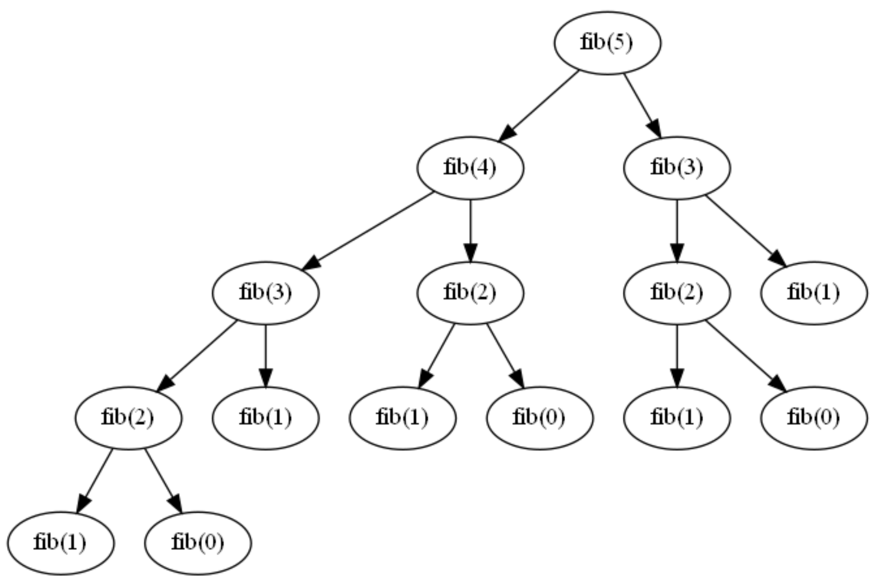

Figure 3: Recursion tree

As you can see in Figure 3, recursion tree is displayed nicely.
Or even better

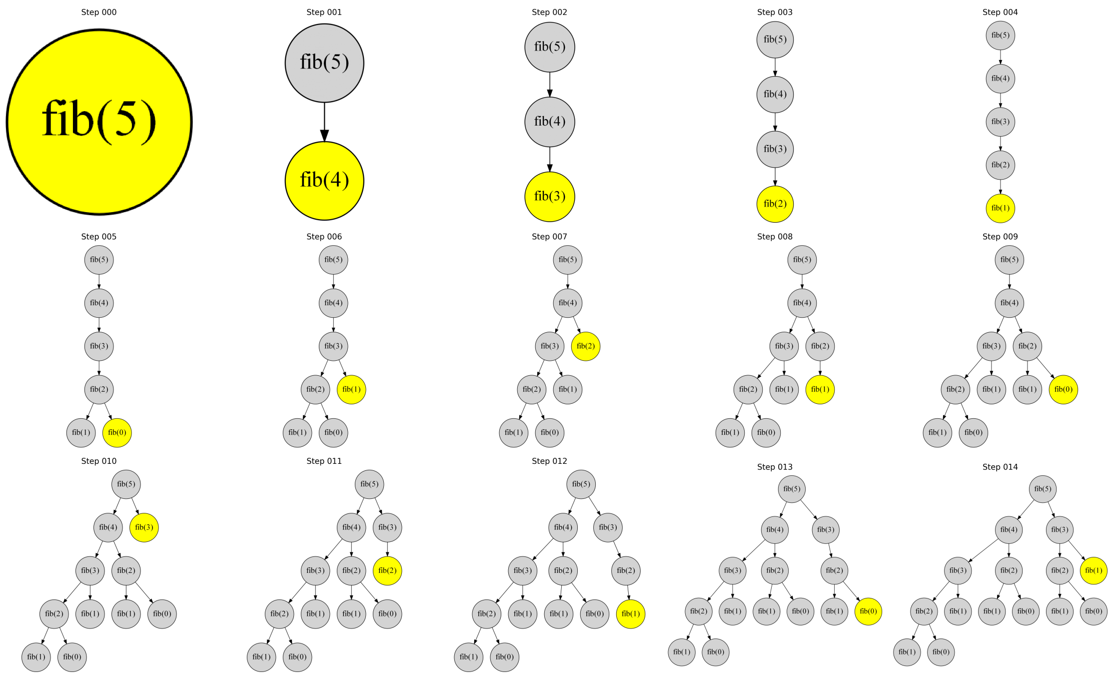

Figure 4: Recursion progress

2.9 Space Complexity

The space complexity is determined by the memory used on the call stack due to recursion. Although the recursion tree contains an exponential number of nodes ($O(\phi^n)$), not all nodes are active simultaneously. The call stack only holds the frames for the active recursive calls along a single path from the root to a leaf.

2.9.1 Call Stack Analysis

Consider the recursion tree for computing $F(n)$. The deepest path in the tree occurs when the recursion follows $F(n) \to F(n-1) \to F(n-2) \to \cdots \to F(0)$, which has a depth of n. At any point, the call stack contains at most n frames, each storing a constant amount of data (e.g., the parameter n and return address).

For example: - When computing $F(n-1)$, the call for $F(n-2)$ is not yet active. - Once $F(n-1)$ is resolved, its stack frame is popped, and $F(n-2)$ is pushed onto the stack.

Thus, the maximum stack depth is n, leading to a space complexity of:

$$O(n).$$

2.9.2 Clarification on Exponential Misconception

The total number of recursive calls is exponential, which might suggest exponential memory usage. However, since only one path of the recursion tree is active at a time, the call stack grows linearly with n, not exponentially.

2.10 JVM Debugger view

```java
import com.sun.jdi.*;
import com.sun.jdi.connect.*;
import com.sun.jdi.event.*;
import com.sun.jdi.request.*;
import java.io.IOException;
import java.util.*;

public class MinimalDebugger {
    private VirtualMachine vm;
    private EventRequestManager eventManager;

    public static void main(String[] args) {
        new MinimalDebugger().debug();
    }

    public void debug() {
        try {
            // 1. Підключаємосядоцільовоїпрограми
            connect();

            // 2. Встановлюємо breakpoint
            setBreakpoint();

            // 3. Запускаємопрограму
            vm.resume();

            // 4. Обробляємоподії
            handleEvents();

        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
        }
    }

    private void connect() throws IOException, IllegalConnectorArgumentsException {
        System.out.println("Connecting to target JVM...");

        VirtualMachineManager vmm = Bootstrap.virtualMachineManager();
        AttachingConnector connector = vmm.attachingConnectors().stream()
            .filter(c -> c.transport().name().equals("dt_socket"))
            .findFirst()
            .orElseThrow(() -> new RuntimeException("Socket connector not found"));

        Map<String, Connector.Argument> args = connector.defaultArguments();
        args.get("hostname").setValue("localhost");
        args.get("port").setValue("5005");

        vm = connector.attach(args);
        eventManager = vm.eventRequestManager();

        System.out.println("Connected to: " + vm.name());
    }

    private void setBreakpoint() {
        // Чекаємозавантаженнякласу
        ClassPrepareRequest classPrepareRequest =
            eventManager.createClassPrepareRequest();
        classPrepareRequest.addClassFilter("FibonacciTarget");
```

```
        classPrepareRequest.enable();
    }

    private void handleEvents() throws InterruptedException {
        EventQueue queue = vm.eventQueue();

        while (true) {
            EventSet eventSet = queue.remove();

            for (Event event : eventSet) {
                if (event instanceof ClassPrepareEvent) {
                    handleClassPrepare((ClassPrepareEvent) event);
                } else if (event instanceof BreakpointEvent) {
                    handleBreakpoint((BreakpointEvent) event);
                } else if (event instanceof VMDeathEvent) {
                    System.out.println("Target VM terminated");
                    return;
                }
            }

            eventSet.resume();
        }
    }

    private void handleClassPrepare(ClassPrepareEvent event) {
        ReferenceType clazz = event.referenceType();
        System.out.println("Class loaded: " + clazz.name());

        // Встановлюємо breakpoint напочатку  main методу
        try {
            Method mainMethod = clazz.methodsByName("main").get(0);
            BreakpointRequest mainBp =
                eventManager.createBreakpointRequest(mainMethod.location());
            mainBp.enable();
            System.out.println("Breakpoint set at main method start");
        } catch (Exception e) {
            System.err.println("Failed to set main breakpoint: " + e.getMessage());
        }

        // Встановлюємо breakpoint вметоді  fibonacci
        try {
            Method fibMethod = clazz.methodsByName("fibonacci").get(0);
            BreakpointRequest fibBp =
                eventManager.createBreakpointRequest(fibMethod.location());
            fibBp.enable();
            System.out.println("Breakpoint set in fibonacci method");
        } catch (Exception e) {
            System.err.println("Failed to set fibonacci breakpoint: " +
                e.getMessage());
        }
    }

    private void handleBreakpoint(BreakpointEvent event) {
        try {
            ThreadReference thread = event.thread();
            StackFrame frame = thread.frame(0);
            Location location = frame.location();
            String methodName = location.method().name();

```

```
            System.out.println("BREAKPOINT in " + methodName + "() at line " + location.lineNumber());

            if ("main".equals(methodName)) {
                System.out.println("=== PROGRAM STARTED ===");
                return;
            }

            if ("fibonacci".equals(methodName)) {
                // Безпечноотримуємопараметр n
                Value nValue = null;
                try {
                    List<LocalVariable> variables = location.method().variables();
                    for (LocalVariable var : variables) {
                        if ("n".equals(var.name())) {
                            nValue = frame.getValue(var);
                            break;
                        }
                    }
                } catch (Exception varError) {
                    System.out.println("Cannot get variable 'n': " + varError.getMessage());
                }

                if (nValue != null) {
                    System.out.println("=== fibonacci(" + nValue + ") ===");
                } else {
                    System.out.println("=== fibonacci(?) ===");
                }

                showStackFrames(thread);
            }

        } catch (Exception e) {
            System.err.println("Error in breakpoint handler: " + e.getClass().getSimpleName() + " - " + e.getMessage());
            e.printStackTrace();
        }
    }

    private void showStackFrames(ThreadReference thread) {
        try {
            List<StackFrame> frames = thread.frames();
            int fibonacciCount = 0;

            System.out.println("Stack depth: " + frames.size());

            for (int i = 0; i < Math.min(frames.size(), 10); i++) {
                StackFrame frame = frames.get(i);
                Location location = frame.location();
                String methodName = location.method().name();

                if ("fibonacci".equals(methodName)) {
                    // Безпечноотримуємозмінну n
                    String nValue = "?";
                    try {
                        List<LocalVariable> variables = location.method().variables();
                        for (LocalVariable var : variables) {
                            if ("n".equals(var.name())) {
```

```
                                Value value = frame.getValue(var);
                                if (value != null) {
                                    nValue = value.toString();
                                }
                                break;
                            }
                        }
                    } catch (Exception e) {
                        // Ігноруємопомилкиотриманнязмінних
                    }

                    System.out.println("  [" + i + "] fibonacci(n=" + nValue + ")");
                    fibonacciCount++;
                } else {
                    System.out.println("  [" + i + "] " + methodName + "()");
                }
            }

            if (frames.size() > 10) {
                System.out.println("  ... and " + (frames.size() - 10) + " more frames");
            }

            System.out.println("Fibonacci calls in stack: " + fibonacciCount);
            System.out.println();

        } catch (Exception e) {
            System.err.println("Error showing stack: " + e.getClass().getSimpleName() + " - " + e.getMessage());
        }
    }
}
```

Listing 4: Minimal Debugger

And target :

```
public class FibonacciTarget {
    public static void main(String[] args){
        System.out.println("Starting Fibonacci calculation...");
        int n = 5;
        long result = fibonacci(n);

        System.out.println("fibonacci(" + n + ") = " + result);
    }

    public static long fibonacci(int n) {
        if (n <= 1) {
            return n;
        }
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Listing 5: Debugger Target

We should compile it with debugging info :

```
javac -g -cp .;%JAVA_HOME%/lib/tools.jar *.java
```

To run app in debug mode we use this:

```
java -cp .;%JAVA_HOME%/lib/tools.jar -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005 FibonacciTarget
```

Then run debugger :

```
java -cp .;%JAVA_HOME%/lib/tools.jar MinimalDebugger
```

Exploring debugger output :

```

C:\temp\fibonacci-article\claude-fibonacci-debugger>java -cp .;C:\server\jdk-22_windows-x64_bin\jdk-22.0.1/lib/tools.jar MinimalDebugger
Connecting to target JVM...
Connected to: Java HotSpot(TM) 64-Bit Server VM
Class loaded: FibonacciTarget
Breakpoint set at main method start
Breakpoint set in fibonacci method
BREAKPOINT in main() at line 3
=== PROGRAM STARTED ===
BREAKPOINT in fibonacci() at line 11
=== fibonacci(5) ===
Stack depth: 2
  [0] fibonacci(n=5)
  [1] main()
Fibonacci calls in stack: 1

BREAKPOINT in fibonacci() at line 11
=== fibonacci(4) ===
Stack depth: 3
  [0] fibonacci(n=4)
  [1] fibonacci(n=5)
  [2] main()
Fibonacci calls in stack: 2

BREAKPOINT in fibonacci() at line 11
=== fibonacci(3) ===
Stack depth: 4
  [0] fibonacci(n=3)
  [1] fibonacci(n=4)
  [2] fibonacci(n=5)
  [3] main()
Fibonacci calls in stack: 3

BREAKPOINT in fibonacci() at line 11
=== fibonacci(2) ===
Stack depth: 5
  [0] fibonacci(n=2)
  [1] fibonacci(n=3)
  [2] fibonacci(n=4)
  [3] fibonacci(n=5)
  [4] main()
Fibonacci calls in stack: 4

BREAKPOINT in fibonacci() at line 11
=== fibonacci(1) ===
Stack depth: 6
  [0] fibonacci(n=1)
  [1] fibonacci(n=2)
  [2] fibonacci(n=3)
```

```
  [3] fibonacci(n=4)
  [4] fibonacci(n=5)
  [5] main()
Fibonacci calls in stack: 5

BREAKPOINT in fibonacci() at line 11
=== fibonacci(0) ===
Stack depth: 6
  [0] fibonacci(n=0)
  [1] fibonacci(n=2)
  [2] fibonacci(n=3)
  [3] fibonacci(n=4)
  [4] fibonacci(n=5)
  [5] main()
Fibonacci calls in stack: 5

BREAKPOINT in fibonacci() at line 11
=== fibonacci(1) ===
Stack depth: 5
  [0] fibonacci(n=1)
  [1] fibonacci(n=3)
  [2] fibonacci(n=4)
  [3] fibonacci(n=5)
  [4] main()
Fibonacci calls in stack: 4

BREAKPOINT in fibonacci() at line 11
=== fibonacci(2) ===
Stack depth: 4
  [0] fibonacci(n=2)
  [1] fibonacci(n=4)
  [2] fibonacci(n=5)
  [3] main()
Fibonacci calls in stack: 3

BREAKPOINT in fibonacci() at line 11
=== fibonacci(1) ===
Stack depth: 5
  [0] fibonacci(n=1)
  [1] fibonacci(n=2)
  [2] fibonacci(n=4)
  [3] fibonacci(n=5)
  [4] main()
Fibonacci calls in stack: 4

BREAKPOINT in fibonacci() at line 11
=== fibonacci(0) ===
Stack depth: 5
  [0] fibonacci(n=0)
  [1] fibonacci(n=2)
  [2] fibonacci(n=4)
  [3] fibonacci(n=5)
  [4] main()
Fibonacci calls in stack: 4

BREAKPOINT in fibonacci() at line 11
=== fibonacci(3) ===
Stack depth: 3
  [0] fibonacci(n=3)
```

```
  [1] fibonacci(n=5)
  [2] main()
Fibonacci calls in stack: 2

BREAKPOINT in fibonacci() at line 11
=== fibonacci(2) ===
Stack depth: 4
  [0] fibonacci(n=2)
  [1] fibonacci(n=3)
  [2] fibonacci(n=5)
  [3] main()
Fibonacci calls in stack: 3

BREAKPOINT in fibonacci() at line 11
=== fibonacci(1) ===
Stack depth: 5
  [0] fibonacci(n=1)
  [1] fibonacci(n=2)
  [2] fibonacci(n=3)
  [3] fibonacci(n=5)
  [4] main()
Fibonacci calls in stack: 4

BREAKPOINT in fibonacci() at line 11
=== fibonacci(0) ===
Stack depth: 5
  [0] fibonacci(n=0)
  [1] fibonacci(n=2)
  [2] fibonacci(n=3)
  [3] fibonacci(n=5)
  [4] main()
Fibonacci calls in stack: 4

BREAKPOINT in fibonacci() at line 11
=== fibonacci(1) ===
Stack depth: 4
  [0] fibonacci(n=1)
  [1] fibonacci(n=3)
  [2] fibonacci(n=5)
  [3] main()
Fibonacci calls in stack: 3

Target VM terminated
```

Listing 6: Call frames in Java Debugger

Debugger class diagram shown here

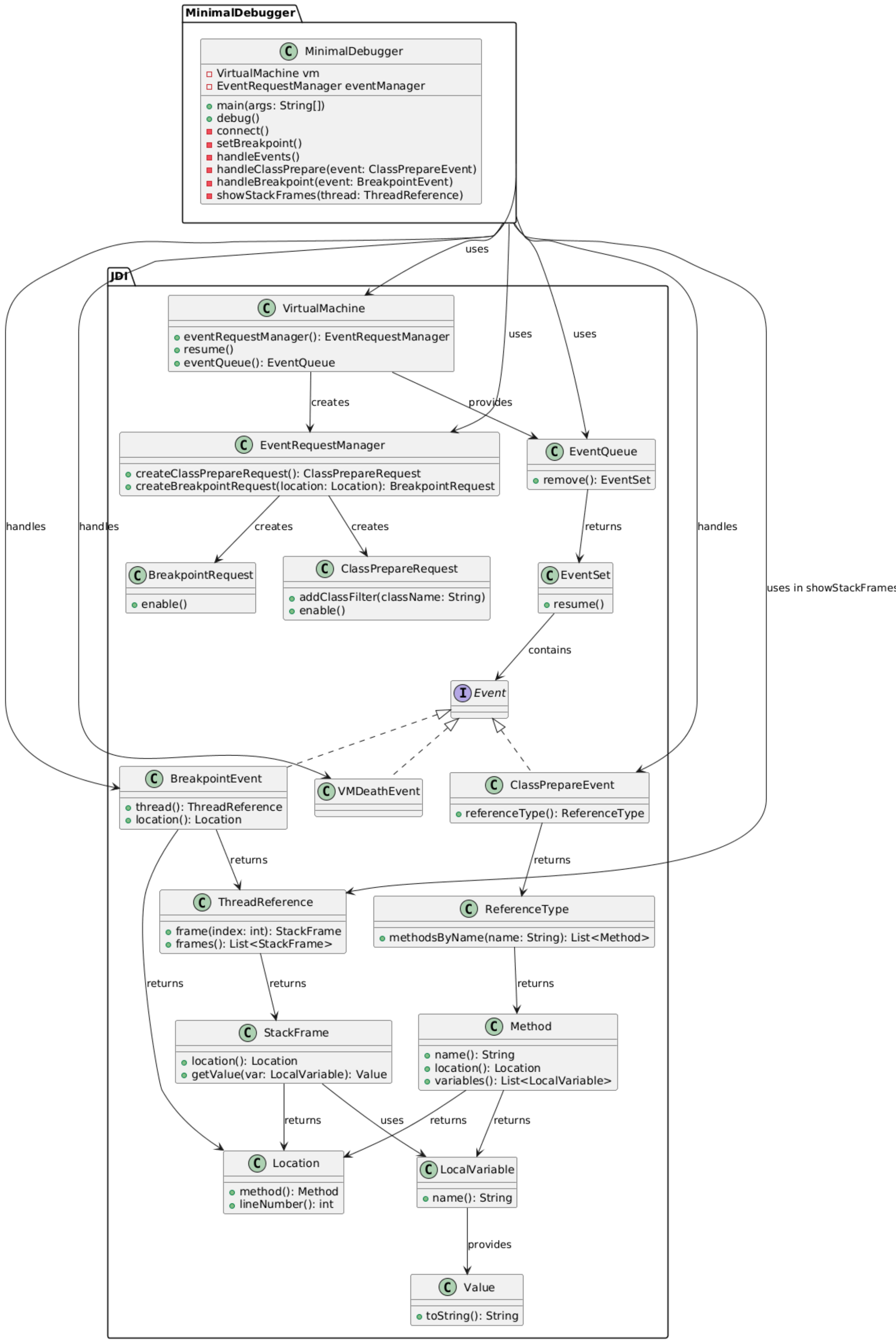

Figure 5: Debugger class diagram

2.11 Conclusion

The recursive Fibonacci algorithm has:

- **Time Complexity**: $O(\phi^n)$, where $\phi \approx 1.618$, due to the exponential number of nodes in the recursion tree.
- **Space Complexity**: $O(n)$, due to the linear depth of the call stack, as only one path of the recursion tree is active at any time.

3 Optimizing Recursion

3.1 Memoization

Memoization stores computed values to avoid redundant calculations.

Time Complexity: $T(n) = \mathcal{O}(n)$
Space Complexity: $M(n) = \mathcal{O}(n)$

```java
import java.util.HashMap;
import java.util.Map;

private static final Map<Integer, Integer> memo = new HashMap<>();
public static int fibMemo(int n) {
 if (n <= 1) return n;
 if (memo.containsKey(n)) return memo.get(n);
 int result = fibMemo(n - 1) + fibMemo(n - 2);
 memo.put(n, result);
 return result;
}
```

Listing 7: Memoized Fibonacci

3.2 Dynamic Programming

Dynamic programming (DP) avoids recursion entirely. We present two variants: array-based and space-optimized.

```java
public static int fibDPArray(int n) {
 if (n <= 0) return 0;
 int[] fib = new int[n + 1];
 fib[0] = 0;
 fib[1] = 1;
 for (int i = 2; i <= n; i++) {
  fib[i] = fib[i - 1] + fib[i - 2];
 }
 return fib[n];
}
```

Listing 8: Array-Based DP Fibonacci

```java
public static int fibDPOptimized(int n) {
 if (n <= 0) return 0;
 if (n == 1) return 1;
 int prev = 0, curr = 1;
 for (int i = 2; i <= n; i++) {
  int next = prev + curr;
  prev = curr;
  curr = next;
 }
 return curr;
}
```

Listing 9: Space-Optimized DP Fibonacci

Time Complexity: $T(n) = \mathcal{O}(n)$
Space Complexity: $M(n) = \mathcal{O}(1)$ (optimized version)

3.3 Linear algebra and Fibonacci

3.3.1 Matrices and Transformations

We begin with the general concepts of matrices as linear transformations and matrix multiplication, then apply these ideas to the Fibonacci sequence, deriving its matrix form and geometric meaning.

Matrices represent linear transformations in vector spaces. A linear transformation $T : \mathbb{R}^m \to \mathbb{R}^n$ preserves vector addition and scalar multiplication:

$$T(\mathbf{u} + \mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v}), \quad T(c\mathbf{u}) = cT(\mathbf{u}).$$

For example, a 2x2 matrix defines a transformation $T : \mathbb{R}^2 \to \mathbb{R}^2$. Geometrically, such transformations can include rotations, scalings, shears, or reflections in the plane.

In the context of sequences like Fibonacci, we will use a specific matrix to model the recurrence as a linear transformation that iteratively evolves a state vector.

Matrix multiplication is defined such that the product AB represents the composition of the linear transformations corresponding to B followed by A. For two matrices A (an $m \times n$ matrix) and B (an $n \times p$ matrix), the element $(AB)_{ij}$ is the dot product of the i-th row of A and the j-th column of B:

$$(AB)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

This definition arises from the need to compose linear transformations. If T_A is the transformation for A and T_B for B, then $T_A(T_B(\mathbf{v})) = (AB)\mathbf{v}$. Improve this def !!!

Geometrically, matrix multiplication corresponds to applying one transformation after another. For example, multiplying rotation matrices composes rotations; scaling followed by shearing distorts shapes accordingly. In sequences, repeated multiplication evolves the state over multiple steps, leading to growth or convergence patterns observable in geometric structures.

Building on linear transformations, we express the Fibonacci recurrence in matrix form. Represent a pair of consecutive Fibonacci numbers as a vector:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}.$$

We seek a matrix $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ such that:

$$A \cdot \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}.$$

Multiplying matrix to vector gives us the system of equations:

$$\begin{cases} aF_n + bF_{n-1} = F_{n+1} \\ cF_n + dF_{n-1} = F_n \end{cases} \tag{1}$$

From the Fibonacci recurrence, we know:

$$F_{n+1} = F_n + F_{n-1}.$$

Comparing this with the first equation from system (**??**), we get:

$$aF_n + bF_{n-1} = F_n + F_{n-1}.$$

For this to hold for all n, the coefficients must match:

$$a = 1, \quad b = 1.$$

For second equation from (**??**), we need:

$$cF_n + dF_{n-1} = F_n.$$

This implies:

$$c = 1, \quad d = 0.$$

Thus, the matrix is:

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

This matrix acts as a linear transformation:

$$A \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + y \\ x \end{bmatrix},$$

which is a shear transformation.

To find later terms, we use matrix powers:

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{2}$$

This is what we were looking for - recurrence relation in matrix form for Fibonacci numbers. We'll visualize (**??**) later because it's fun.

3.3.2 Algorithm implementation in Java

```
public static long fibonacciMatrix(int n) {
 if (n <= 0) return 0;
 long[][] matrix = {{1, 1}, {1, 0}};
 long[][] result = matrixPower(matrix, n - 1);
 return result[0][0];
}

static long[][] matrixPower(long[][] matrix, int n) {
 int row = matrix.length;
 long[][] result = {{1, 0}, {0, 1}};
 while (n > 0) {
  if (n % 2 == 1) result = matrixMultiply(result, matrix);
  matrix = matrixMultiply(matrix, matrix);
  n /= 2;
 }
 return result;
}

static long[][] matrixMultiply(long[][] a, long[][] b) {
 long[][] result = new long[2][2];
 for (int i = 0; i < 2; i++)
 for (int j = 0; j < 2; j++)
 for (int k = 0; k < 2; k++)
 result[i][j] += a[i][k] * b[k][j];
 return result;
}
```

Listing 10: Matrix Exponentiation for Fibonacci

3.3.3 Time and space complexity

Time Complexity: $T(n) = \mathcal{O}(\log n)$
Space Complexity: $M(n) = \mathcal{O}(1)$

3.4 Computer Algebra Systems

Using Maple, we can compute Fibonacci numbers efficiently:

```
with(combinat, fibonacci);
fibonacci_numbers := seq(fibonacci(i), i = 0 .. 100000):
f := fopen("fibonacci_numbers.txt", WRITE):
for num in fibonacci_numbers do
fprintf(f, "%a\n", num);
end do:
fclose(f);
```

Listing 11: Fibonacci in Maple

Run with:

```
cmaple -q fibonacci.mpl >out.log
```

Listing 12: Running Maple Script

3.5 Binet formula in real life

The Binet formula provides a closed-form expression for the n-th Fibonacci number:

$$F(n) = \frac{\phi^n - \psi^n}{\sqrt{5}},$$

where $\phi = \frac{1+\sqrt{5}}{2}$ (the golden ratio) and $\psi = \frac{1-\sqrt{5}}{2}$. Since it directly computes $F(n)$ without iterative steps, it has a theoretical time complexity of $O(1)$ for a single computation, assuming arithmetic operations (like exponentiation) are constant-time. However, despite this apparent efficiency, the Binet formula is impractical for large n in computational practice for several reasons.

1. **Numerical Instability with Floating-Point Arithmetic**: For large n, ϕ^n grows exponentially, while ψ^n (with $|\psi| < 1$) becomes extremely small. Their difference, $\phi^n - \psi^n$, involves subtracting two nearly equal values, leading to significant round-off errors in floating-point arithmetic (e.g., `double` in Java or C++). This causes inaccuracies, as seen when computing $F(71)$, where floating-point results may deviate (e.g., 308061521170131 instead of the correct 308061521170129).

2. **High Precision Requirements with Arbitrary-Precision Arithmetic**: To mitigate floating-point issues, arbitrary-precision arithmetic (e.g., `BigDecimal` in Java) can be used. However, computing ϕ^n, ψ^n, and $\sqrt{5}$ requires high precision (e.g., hundreds of decimal places for $n \approx 100$) to ensure the result is an exact integer after division by $\sqrt{5}$. This significantly increases computational cost, as each operation (exponentiation, division) scales with the number of digits, making the effective time complexity much worse than $O(1)$.

3. **Computational Overhead of Irrational Numbers**: The Binet formula involves irrational numbers (ϕ, ψ, $\sqrt{5}$), which are computationally expensive to handle with high precision. In contrast, iterative methods using integer arithmetic (e.g., with `BigInteger`) involve only simple additions, which are faster and inherently exact for Fibonacci numbers, as they are integers by definition.

4. **Scalability Issues for Large n**: For very large n (e.g., $n = 1000$), the precision required for $\sqrt{5}$ and ϕ^n grows proportionally to n, as the number of digits in $F(n)$ is approximately $n \cdot \log_{10}(\phi)$. This makes Binet's formula slower than iterative or matrix-based methods, which have logarithmic complexity ($O(\log n)$ with fast exponentiation) but operate on integers.

5. **Implementation Complexity**: Implementing Binet's formula requires careful management of precision and rounding to ensure integer results, which adds complexity compared to the straightforward iterative approach (e.g., $F(n) = F(n-1) + F(n-2)$). The latter is simpler to code and debug, as it avoids dealing with floating-point or arbitrary-precision libraries.

In conclusion, while the Binet formula is theoretically $O(1)$, its practical performance is hindered by numerical instability, high precision requirements, and computational overhead for irrational numbers. Iterative methods, or matrix exponentiation methods with $O(\log n)$ complexity, are preferred in practice due to their simplicity, exactness, and efficiency, especially for large n. For example, computing $F(71)$ iteratively with `BigInteger` is faster and guarantees the correct result (308061521170129) without precision management.

3.6 Conclusion

We explored Fibonacci computation methods, from naive recursion ($\mathcal{O}(2^n)$) to matrix exponentiation ($\mathcal{O}(\log n)$). Below is a comparison:

Method	**Time Complexity**	**Space Complexity**
Naive Recursion	$\mathcal{O}(2^n)$	$\mathcal{O}(n)$
Memoization	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Dynamic Programming	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Binet's Formula	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Matrix Exponentiation	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

Table 4: Complexity Comparison of Fibonacci Methods

For large n, matrix exponentiation or BigDecimal-based DP are recommended. Future exploration could include fast doubling algorithms or parallel computation.

4 Limitations of Primitive Types for Large Fibonacci Numbers

4.1 Showcase : from int to double

```java
import java.math.BigDecimal;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

// Strategy interface for exact addition
interface AdditionStrategy<T extends Number> {
    T add(T a, T b) throws ArithmeticException;
}

// Strategy for Integer
class IntegerAdditionStrategy implements AdditionStrategy<Integer> {
    @Override
    public Integer add(Integer a, Integer b) throws ArithmeticException {
        return Math.addExact(a, b);
    }
}

// Strategy for Long
class LongAdditionStrategy implements AdditionStrategy<Long> {
    @Override
    public Long add(Long a, Long b) throws ArithmeticException {
        return Math.addExact(a, b);
    }
}


// Strategy for Double
// Note: Unlike int/long, Java does not provide Math.addExact for double/float.
// This is because floating-point arithmetic (IEEE 754) does not throw overflow
// exceptions: instead, results silently become Infinity, -Infinity, or NaN.
// If strict overflow detection is required, it must be implemented manually
// (e.g., by checking Double.isInfinite / Double.isNaN), or by using BigDecimal
// for arbitrary precision arithmetic.

class FloatAdditionStrategy implements AdditionStrategy<Float> {
    @Override
    public Float add(Float a, Float b) throws ArithmeticException {
        float result = a + b;

        // Check for positive or negative infinity (overflow)
        if (Float.isInfinite(result)) {
            throw new ArithmeticException("Float overflow: " + a + " + " + b);
        }

        // Check for Not-a-Number result
        if (Float.isNaN(result)) {
            throw new ArithmeticException("Result is NaN: " + a + " + " + b);
        }

        return result;
    }
}

```

```java
class DoubleAdditionStrategy implements AdditionStrategy<Double> {
    @Override
    public Double add(Double a, Double b) throws ArithmeticException {
        double result = a + b;
        if (Double.isInfinite(result)) {
            throw new ArithmeticException("Double overflow: " + a + " + " + b);
        }
        if (Double.isNaN(result)) {
            throw new ArithmeticException("Result is NaN: " + a + " + " + b);
        }
        return result;
    }
}

class FibonacciResult<T extends Number> {
    private final int index;
    private final T value;

    private FibonacciResult(int index, T value) {
        this.index = index;
        this.value = value;
    }


    public static <T extends Number> FibonacciResult<T> of(int index, T value) {
        return new FibonacciResult<>(index, value);
    }

    public Class<T> getType() {
        return (Class<T>) value.getClass();
    }

    public int getIndex() {
        return index;
    }

    public T getValue() {
        return value;
    }

    public String toString() {
        return switch (value) {
            case Integer i -> Integer.toString(i);
            case Long l -> Long.toString(l);
            case Float f -> BigDecimal.valueOf(f).toPlainString();
            case Double d -> BigDecimal.valueOf(d).toPlainString();

            default -> value.toString();
        };
    }
}

@SuppressWarnings("all")
public class LargestExactFibonacci {
    public static void main(String[] args) {
        Map<Class<? extends Number>, AdditionStrategy<?>> strategies = new
            HashMap<>();
        strategies.put(Integer.class, new IntegerAdditionStrategy());
        strategies.put(Long.class, new LongAdditionStrategy());
```

```
        strategies.put(Float.class, new FloatAdditionStrategy());
        strategies.put(Double.class, new DoubleAdditionStrategy());


        FibonacciResult<Integer> resultInt = findLargestExactFibonacci(0, 1,
                (AdditionStrategy<Integer>) strategies.get(Integer.class));
        FibonacciResult<Long> resultLong = findLargestExactFibonacci(0L, 1L,
                (AdditionStrategy<Long>) strategies.get(Long.class));
        FibonacciResult<Float> resultFloat = findLargestExactFibonacci(0.0F, 1.0F,
                (AdditionStrategy<Float>) strategies.get(Float.class));
        FibonacciResult<Double> resultDouble = findLargestExactFibonacci(0.0D, 1.0D,
                (AdditionStrategy<Double>) strategies.get(Double.class));

        for (FibonacciResult<?> entry : List.of(resultInt, resultLong, resultFloat,
            resultDouble)) {
            System.out.printf("%s Fibonacci (%s): F(%s) = %s%n",
                    entry.getType().getSimpleName(), entry.getIndex(),
                        entry.getIndex(), entry);
        }
    }
    private static <T extends Number> FibonacciResult<T> findLargestExactFibonacci(
            T prev, T current, AdditionStrategy<T> strategy) {
        int index = 1;

        while (true) {
            try {
                T next = strategy.add(prev, current);
                prev = current;
                current = next;
                index++;
            } catch (ArithmeticException e) {
                return FibonacciResult.of(index, current);
            }
        }
    }
}
```

4.2 When double goes wild

4.2.1 $0.1 + 0.2 \neq 0.3$

In Java, when running the program

```
public class A {
 public static void main(String[] args) {
  double x = 0.1;
  double y = 0.2;
  double z = x + y;
  System.out.println(z);
 }
}
```

the output is

```
0.30000000000000004
```

instead of exactly `0.3`.

If you think Java is broken, try it in JavaScript or Python
JS:

```
const x = 0.1;
const y = 0.2;
const z = x + y;
console.log(z);
```

Python:

```
x = 0.1
y = 0.2
z = x + y
print(z)
```

Output :

```
C:\temp\>java A
0.30000000000000004

C:\temp\>node a.js
0.30000000000000004

C:\temp\>python a.py
0.30000000000000004
```

This behavior is not a bug, but a direct consequence of the *IEEE 754 double-precision floating-point format*.

4.2.2 IEEE 754 Representation

A Java `double` uses 64 bits:

- 1 bit for the *sign*,
- 11 bits for the *exponent*,
- 52 bits for the *mantissa* (fraction).

The value is computed as

$$(-1)^{\text{sign}} \times 1.\text{mantissa} \times 2^{(\text{exponent}-1023)}.$$

4.2.3 Calculate as machines

The decimal fraction $0.1 = \frac{1}{10}$ has no finite binary expansion. In base 2 it becomes

$$0.1_{10} = 0.00011001100110011\ldots_2$$

with repeating pattern `0011`. Since only 52 bits are available for the mantissa, this value is stored approximately as

$$0.10000000000000000555\ldots$$

Similarly,

$$0.2 \approx 0.2000000000000000111, \quad 0.3 \approx 0.2999999999999999889.$$

Therefore,

$$0.1 + 0.2 \approx 0.3000000000000000444,$$

which is printed as `0.30000000000000004`.
The decimal system can exactly represent fractions like 1/10 or 1/5, but binary cannot. Binary floating-point can exactly represent only fractions whose denominators are powers of two (e.g., 1/2, 1/4, 1/8). Numbers like 1/10 or 1/5 become infinite binary fractions, which must be rounded.

4.2.4 Conclusion

The result `0.30000000000000004` is not an error, but the inevitable effect of representing decimal fractions in binary with limited precision. Understanding IEEE 754 is essential for writing correct numerical programs.

4.3 Handling Large Numbers with BigDecimal

The `int` and `long` types in Java overflow for large Fibonacci numbers (e.g., F_{93} exceeds `long`). We use `BigDecimal` for arbitrary-precision arithmetic to address this limitation.

```java
import java.math.BigDecimal;

public static BigDecimal fibBigDecimal(int n) {
    if (n <= 0) return BigDecimal.ZERO;
    if (n == 1) return BigDecimal.ONE;

    BigDecimal prev = BigDecimal.ZERO;
    BigDecimal curr = BigDecimal.ONE;
    for (int i = 2; i <= n; i++) {
        BigDecimal next = prev.add(curr);
        prev = curr;
        curr = next;
    }
    return curr;
}
```

Listing 13: Fibonacci with BigDecimal

Time Complexity: $T(n) = \mathcal{O}(n \cdot M)$, where M is the cost of BigDecimal operations.
Space Complexity: $M(n) = \mathcal{O}(1)$

5 Real-World Applications

5.1 Fibonacci heaps for Dijkstra's algorithm optimization

bla-bla-bla

5.2 Fibonacci retracement levels in stock market analysis

bla-bla-bla

5.3 Some of pseudorandom number generators

bla-bla-bla

References

[1] Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley.

[2] MapleSoft. (2025). Fibonacci Function Documentation. `https://www.maplesoft.com/support/help/Maple/view.aspx?path=combinat/fibonacci`.

[3] Wikipedia. (2025). Fibonacci Number. `https://en.wikipedia.org/wiki/Fibonacci_number`.

[4] SciPy docs `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html`