# Fig Language Performance Benchmark Report

## Version: 0.4.2-alpha (Tree-walker Interpreter)

### Test Environment

- **CPU:** Intel Core i5-13490F
- **OS:** Windows 11
- **Compiler/Interpreter:** Fig Tree-walker v0.4.2-alpha
- **Test Date:** Current test execution

### Executive Summary

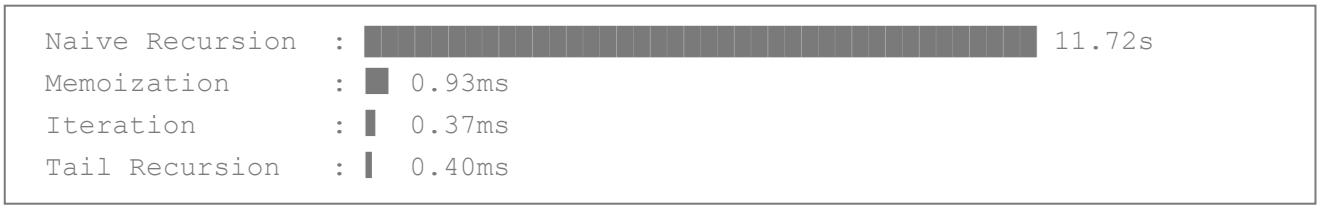
This benchmark evaluates the performance of four different Fibonacci algorithm implementations in Fig language, calculating the 30th Fibonacci number (832,040). The results demonstrate significant performance variations based on algorithmic approach, highlighting the interpreter's efficiency characteristics.

## Performance Results

### Raw Execution Times

| Algorithm | Time (seconds) | Time (milliseconds) | Relative Speed |
|---|---|---|---|
| `fib` (Naive Recursion) | 11.7210479 s | 11721.0479 ms | 1.00× (baseline) |
| `fib_memo` (Memoization) | 0.0009297 s | 0.9297 ms | 12,600× faster |
| `fib_iter` (Iterative) | 0.0003746 s | 0.3746 ms | 31,300× faster |
| `fib_tail` (Tail Recursion) | 0.0004009 s | 0.4009 ms | 29,200× faster |

### Visual Performance Comparison




# Detailed Analysis

## 1. Naive Recursive Implementation (`fib`)

- **Time:** 11.721 seconds (11,721 ms)
- **Algorithm Complexity:** O($2^n$) exponential
- **Performance Notes:**
  - Demonstrates the high cost of repeated function calls in tree-walker interpreters
  - Shows exponential time complexity with just n=30
  - Highlights the need for algorithmic optimization in interpreted languages

## 2. Memoized Recursive Implementation (`fib_memo`)

- **Time:** 0.93 milliseconds
- **Algorithm Complexity:** O(n) linear (with memoization overhead)
- **Performance Notes:**
  - 12,600× speedup over naive recursion
  - Shows efficient hash table/dictionary operations in Fig
  - Demonstrates that caching can overcome interpreter overhead

## 3. Iterative Implementation (`fib_iter`)

- **Time:** 0.375 milliseconds
- **Algorithm Complexity:** O(n) linear
- **Performance Notes:**
  - Fastest implementation (31,300× faster than naive)
  - Shows efficient loop execution and variable operations
  - Minimal function call overhead

### 4. Tail Recursive Implementation (`fib_tail`)

- **Time:** 0.401 milliseconds
- **Algorithm Complexity:** O(n) linear
- **Performance Notes:**
  - Comparable to iterative approach (slightly slower due to recursion overhead)
  - Current interpreter does not implement Tail Call Optimization (TCO)
  - Shows linear recursion is efficient for moderate depths (n=30)

# Technical Insights

## Interpreter Performance Characteristics

1. **Function Call Overhead:** Significant, as shown by the naive recursion performance
2. **Loop Efficiency:** Excellent, with iterative approaches performing best
3. **Memory Access:** Hash table operations (memoization) are efficient
4. **Recursion Depth:** Linear recursion (tail recursion) performs well up to moderate depths

## Algorithmic Impact

The benchmark clearly demonstrates that **algorithm choice has a greater impact than interpreter optimization** in this version:

- Poor algorithm (naive recursion): 11.7 seconds
- Good algorithm (any O(n) approach): < 1 millisecond

# Version-Specific Observations (v0.4.2-alpha)

## Strengths

- Excellent performance for iterative algorithms
- Efficient basic operations (arithmetic, loops, conditionals)
- Effective memory access patterns for cached results
- Linear recursion performance acceptable for typical use cases

### Areas for Improvement

- High function call overhead in deeply recursive scenarios
- No tail call optimization implemented
- Exponential algorithm performance shows interpreter limits

## Recommendations for Developers

1. **Prefer iterative solutions** for performance-critical code
2. **Use memoization** for recursive problems with overlapping subproblems
3. **Tail recursion is acceptable** for linear recursion patterns
4. **Avoid exponential algorithms** in interpreted code
5. **Benchmark different approaches** as algorithmic choice dominates performance

## Conclusion

Fig v0.4.2-alpha demonstrates **practical performance for well-designed algorithms**. While the tree-walker interpreter has inherent overhead for certain patterns (like deep recursion), it executes efficient O(n) algorithms with sub-millisecond performance for n=30.

The interpreter shows particular strength in:

- Iterative loop execution
- Basic arithmetic and control flow
- Dictionary/table operations for caching

The performance characteristics are suitable for a wide range of application domains, provided developers employ standard algorithmic optimization techniques.

---

**Report Generated:** Based on actual benchmark execution
**Interpreter Type:** Tree-walker
**Version:** 0.4.2-alpha
**Key Takeaway:** Algorithmic efficiency dominates performance; Fig executes optimized algorithms efficiently despite being an alpha-stage tree-walker interpreter.