Programming Assignment

Performance Comparison: Recursive Fibonacci

Amir Mirzaeinia

CSCE @ UNT

Outline

Assignment Overview

Objective

Compare execution performance between low-level C and high-level Python programming languages using a recursive Fibonacci implementation.

What You'll Learn

- Performance differences between compiled and interpreted languages
- Trade-offs between development speed and execution speed
- When to choose C vs Python for different tasks
- Impact of language-level abstractions on performance

Assignment Tasks

Part 1: Implementation

1. Implement Fibonacci function **recursively** in C
2. Implement Fibonacci function **recursively** in Python
3. Calculate `fibonacci(50)`
4. Measure execution time for both

Part 2: Performance Analysis

1. Compare C vs Python performance
2. Document your findings in a report

What is Recursion?

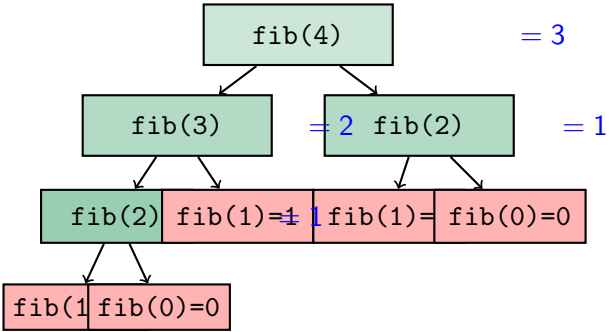
Definition

Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller, similar subproblems.

Key Components

1. **Base Case**: The simplest case that can be solved directly (stops recursion)
2. **Recursive Case**: Calls the function with a simpler input

Recursion: The Concept



Notice: Many function calls! fibonacci(2) called twice, fibonacci(1) called three times!

Fibonacci: Mathematical Definition

What is Fibonacci Sequence?

The Fibonacci sequence is a series where each number is the sum of the two preceding ones: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Iterative View

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(2) &= 0 + 1 = 1 \\
F(3) &= 1 + 1 = 2 \\
F(4) &= 1 + 2 = 3 \\
F(5) &= 2 + 3 = 5
\end{aligned}
$$

Recursive Definition

$$
F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}
$$

Example: $F(5) = F(4) + F(3)$

Examples

- $F(0) = 0$, $F(1) = 1$
- $F(5) = 5$, $F(10) = 55$
- $F(20) = 6,765$, $F(30) = 832,040$

Step-by-Step: Implementing Recursive Fibonacci

Step 1: Identify Base Cases

The **simplest cases** that don't require recursion:

- For Fibonacci: When $n = 0$, answer is 0
- When $n = 1$, answer is 1
- These prevent infinite recursion

Step 2: Identify Recursive Case

How to **reduce the problem** to simpler versions:

- For Fibonacci: $F(n) = F(n-1) + F(n-2)$
- Each call handles two smaller subproblems

Template Pattern

```
function recursive_function(input):
    if (base_case_condition_1):
        return base_case_value_1
    if (base_case_condition_2):
        return base_case_value_2
    else:
        return combine(recursive_function(smaller_1),
                       recursive_function(smaller_2))
```

Implementation: C Language

```c
unsigned long long fibonacci(int n) {
    // Step 1: Base cases - stop recursion
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    // Step 2: Recursive case - sum of two previous numbers
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

How It Works

1. `fibonacci(5)` calls `fibonacci(4)` and `fibonacci(3)`
2. Each of those calls itself recursively
3. Continues until reaching base cases (`fibonacci(0)` or `fibonacci(1)`)
4. Results bubble back up: $F(5) = F(4) + F(3) = 3 + 2 = 5$

Implementation: Python Language

```python
def fibonacci(n):
    # Step 1: Base cases - stop recursion
    if n == 0:
        return 0
    if n == 1:
        return 1
    # Step 2: Recursive case - sum of two previous numbers
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Key Differences from C

- No type declarations needed (dynamic typing)
- Python automatically handles arbitrarily large integers
- Syntax is more concise
- **But**: Function call overhead is MUCH higher!
- **Warning**: Recursive Fibonacci is exponentially slow in both languages!

Tracing Recursive Execution

Example: fibonacci(4)

1. Call `fibonacci(4)`
   - Not base case, so call `fibonacci(3)` and `fibonacci(2)`
2. Call `fibonacci(3)`
   - Call `fibonacci(2)` and `fibonacci(1)`
3. Call `fibonacci(2)`
   - Call `fibonacci(1)` and `fibonacci(0)`
4. `fibonacci(1)` returns 1, `fibonacci(0)` returns 0
5. `fibonacci(2)` gets $1 + 0 = 1$
6. `fibonacci(3)` gets fibonacci(2) + fibonacci$(1) = 1 + 1 = 2$
7. `fibonacci(4)` gets fibonacci(3) + fibonacci$(2) = 2 + 1 = 3$

Final Result: 3

Common Recursion Mistakes

Mistake 1: Missing or Wrong Base Case

- **Problem**: Infinite recursion, stack overflow
- **Example**: Forgetting `if (n == 0 || n == 1) return 1;`
- **Result**: Program crashes with "stack overflow" error

Mistake 2: Not Making Problem Smaller

- **Problem**: Infinite recursion
- **Example**: Calling `factorial(n)` instead of `factorial(n-1)`
- **Result**: Never reaches base case

Mistake 3: Integer Overflow (C only)

- **Problem**: Result too large for data type
- **Example**: Using `int` instead of `unsigned long long`
- **Result**: Wrong answer (overflow wraps around)

Starter Code: C Implementation

```c
#include <stdio.h>
#include <time.h>
// TODO: Implement recursive fibonacci function
unsigned long long fibonacci(int n) {
    // Your code here
    // Base case 1: if n == 0, return ?,  Base case 2: if n == 1,
       return ?
    // Recursive case: return ?
}
```

Starter Code: C Implementation (Continued)

```c
int main() {
    int n = 50;
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    unsigned long long result = fibonacci(n);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Fibonacci(%d) = %llu\n", n, result);
    printf("Time taken: %f seconds\n", cpu_time_used);
    return 0;
}
```

Starter Code: C Implementation (Continued)

Compilation Instructions

To compile:

```
gcc -o factorial factorial.c
```

To run:

```
./factorial
```

Tips for C Implementation

- Use `unsigned long long` for the return type
- Base case 1: `if (n == 0) return 0;`
- Base case 2: `if (n == 1) return 1;`
- Recursive case: `return fibonacci(n - 1) + fibonacci(n - 2);`
- Use `clock()` from `time.h` for timing
- **Warning**: This will be SLOW for n=50! That's the point!

Starter Code: Python Implementation

```python
import time
# TODO: Implement recursive fibonacci function
def fibonacci(n):
    """
    Calculate nth Fibonacci number recursively
    Args:
        n: Non-negative integer
    Returns:
        nth Fibonacci number
    """
    # Your code here
    # Base case 1: if n == 0, return ?
    # Base case 2: if n == 1, return ?
    # Recursive case: return ?
    pass
```

Starter Code: Python Implementation

```python
if __name__ == "__main__":
    n = 50

    start_time = time.time()
    result = fibonacci(n)
    end_time = time.time()

    execution_time = end_time - start_time

    print(f"Fibonacci({n}) = {result}")
    print(f"Time taken: {execution_time:.6f} seconds")
```

Starter Code: Python Implementation (Continued)

Execution Instructions

To run:

```
python3 factorial.py
```

Tips for Python Implementation

- Python handles large integers automatically
- Base case 1: `if n == 0: return 0`
- Base case 2: `if n == 1: return 1`
- Recursive case: `return fibonacci(n - 1) + fibonacci(n - 2)`
- Use `time.time()` for timing measurements
- No recursion limit issues for n=50 (not deep enough)
- **Warning**: This will be VERY SLOW! Expect several seconds/minutes/hours (depends on your device)!

Important Notes

WARNING About This Implementation

This recursive approach is **intentionally inefficient** for computing Fibonacci. There are much more efficient methods:

- **Iterative approach**: O(n) time, O(1) space - MUCH faster!
- ...

You will learn these optimized approaches in your **Algorithm Design course**.

Purpose of This Assignment

The goal is **NOT** to write the best Fibonacci algorithm, but rather to:

1. Observe **performance differences** between C and Python
2. Understand the impact of **compiled vs interpreted** execution
3. Appreciate the **trade-offs** in language selection
4. **Experience exponential time complexity** firsthand!

Why Recursive? Why Fibonacci?

Why Recursive Implementation?

- Tests function call overhead
- Fibonacci makes MANY repeated calls
- More expensive in Python (dynamic typing, interpreter overhead)
- Magnifies performance differences dramatically
- Shows exponential time complexity

Why Fibonacci?

- Simple algorithm everyone can understand
- Exponential number of recursive calls (2^n)
- Performance differences are huge!
- Demonstrates one difference between higher level and lower level language

Fun Fact

fibonacci(40) makes over **330 million function calls**!
fibonacci(50) would make over **40 billion calls**! (Too slow to run!)

Expected Results

Performance Expectations

You should observe:

- **C will be significantly faster** (10-100x faster typical)
- C's compiled nature eliminates interpretation overhead
- Python's dynamic typing adds runtime checks
- Function call overhead is much higher in Python

Example Timing (Approximate)

Language	**Typical Time for fibonacci(50)**
C (compiled with -O2)	$\sim$ 0.3 - 1.0 seconds
Python (interpreted)	$\sim$ 30 - 60 seconds(minutes)
Speed Difference	**50-100x faster in C!**

What to Submit

Required Files

1. `fibonacci.c` - Your C implementation
2. `fibonacci.py` - Your Python implementation
3. `report.pdf` - Your analysis report (2-3 pages)

Report Contents

Your report should include:

- **Implementation description**: Brief explanation of your code
- **Testing methodology**: How you measured performance
- **Results**: Table/graph showing timing data (multiple runs!)
- **Analysis**: Why C is faster, discuss exponential growth
- **Trade-offs**: When would you choose Python despite being slower?
- **Verification**: Show both produce correct result (fibonacci(50) = 102,334,155)

Grading Rubric

Component	Points
C implementation (correct & recursive)	25
Python implementation (correct & recursive)	25
Performance measurements (accurate timing)	20
Analysis & comparison (insightful discussion)	20
Code quality & documentation	10
Total	**100**

Due Date

Check your course syllabus or LMS for the exact due date and submission instructions.

Tips for Success

Debugging Tips

- Test with small values first (fibonacci(5), fibonacci(10))
- Verify correctness: fibonacci(5) = 5, fibonacci(10) = 55, fibonacci(20) = 6,765
- Use `printf` (C) or `print` (Python) to trace recursive calls
- **Don't test with large values immediately!** fibonacci(50) takes time!
- If base cases are wrong, you'll get incorrect results or infinite recursion

Performance Measurement Tips

- Run each program multiple times (3-5 iterations for fibonacci(50))
- Calculate average time to reduce measurement noise
- Close other applications to minimize system interference
- Use the same hardware for both measurements
- **Be patient!** fibonacci(50) in Python takes 30-60 seconds/Minutes/Hours(depending on your device)
- Consider testing smaller values first (fibonacci(35) is faster)

Common Pitfalls to Avoid

Common Mistakes

1. **Wrong base cases**: Must handle BOTH n=0 AND n=1
   - fibonacci(0) = 0, fibonacci(1) = 1 (not both 1!)
2. **Missing base case**: Causes infinite recursion and stack overflow
3. **Testing with n=50**: Too slow! Use n=40 or smaller
   - fibonacci(50) in Python could take hours!
4. **Only running once**: fibonacci(40) times vary, need multiple runs
5. **Comparing debug C vs Python**: Always compile C with optimizations
   - Use: `gcc -O2 -o fibonacci fibonacci.c`

Questions?

Contact your instructor or TA during office hours

Good luck and have fun with your assignment!