

Factorials in Julia: Recursive vs Iterative vs Built-in

This handout introduces three ways to compute factorials in Julia and compares their efficiency:

1. Recursive function (matches the mathematical definition).
2. Iterative loop (practical, efficient, avoids recursion depth).
3. Julia's built-in `factorial` (highly optimized).

We also compare how different integer types (`Int64` vs `BigInt`) affect results.

Recursive version

```
function my_factorial_recursive(n::Integer)
    if n < 0
        throw(DomainError(n, "factorial is not defined for negative integers"))
    elseif n ≤ 1
        return 1
    else
        return n * my_factorial_recursive(n - 1)
    end
end
```

Iterative version

```
function my_factorial_iterative(n::Integer)
    if n < 0
        throw(DomainError(n, "factorial is not defined for negative integers"))
    end
    result = 1
    for i in 2:n
        result *= i
    end
    return result
end
```

Built-in factorial

Julia provides:

```
factorial(20)           # works for Int up to 20
factorial(BigInt(50))  # arbitrary precision with BigInt
```

- `factorial(Int)` is defined only up to 20.
- `factorial(BigInt)` supports arbitrary `n` (limited by memory).

Benchmarking

```
using BenchmarkTools

@btime my_factorial_recursive(20);
@btime my_factorial_iterative(20);
@btime factorial(BigInt(20));
```

- Recursive is elegant but slow.
- Iterative is faster and safe for large `n`.
- Built-in is highly optimized.

Safe Ranges for Factorials

- `Int64` overflows after 20!
- `BigInt` supports arbitrarily large factorials (limited only by memory).
- Recursive implementations fail around ~10,000 due to stack overflow.

Key Takeaways

- **Recursion** mirrors math but is inefficient for large `n`.
- **Iteration** is efficient and safe.
- **Built-in** is best in practice.
- Understanding integer types is critical in computational physics.