Basic Operations Documentation

Course: Algorithms

Team members: Meirambek, Ernar, Yestay

1. Project Overview

The project implements the four fundamental arithmetic operations (addition, subtraction, multiplication, division) for arbitrarily large integers using only arrays/lists of digits. Two versions with identical algorithms were developed:

- Python implementation (educational and easy to debug)
- C++ implementation (high-performance)

Numbers are stored in base-10, least significant digit first.

2. Implemented Operations and Algorithms

Operation	Algorithm	Time Complexity
Addition	Schoolbook addition with carry propagation	**O(n)**
Subtraction	Schoolbook subtraction with borrow (a ≥ b)	**O(n)**
Multiplication	Karatsuba algorithm (recursive, divide-and-conquer)	**O(n^{log_2 3}) ≈ O(n^{1.585})**
	Automatic fallback to O(n²) for small inputs	
Division	Classic long division (digit-by-digit)	**O(n²)**

where n is the number of digits of the larger operand.

3. Correctness Tests

Both implementations correctly compute:

```
text

123 + 456 = 579
123 × 456 = 56088
123 ÷ 3   = 41 remainder 0
```

and give results identical to Python's built-in arbitrary-precision integers on 100-digit test cases.

4. Performance Results (100-digit numbers, 100 operations each)

Operation	Python (seconds)	C++ (milliseconds)	C++ speedup
Addition	0.0031 s	0.0407 ms	~76×
Subtraction	0.0020 s	0.0181 ms	~112×
Multiplication	1.315 s	0.0476 ms	~27 650×

5. Key Observations

- Identical algorithms show dramatically different real-world performance due to interpretation vs compilation.
- Karatsuba becomes noticeably faster than the simple $O(n^2)$ method starting from approximately 50–70 digits.
- The current division algorithm is the primary performance bottleneck.

6. Future Work

1. Change internal representation from base-10 to base-10^9 (32/64-bit limbs) → expected 5–15× speedup for all operations
2. Replace naive division with a fast division algorithm (Newton-Raphson or Burnikel-Ziegler) → bring division time close to multiplication time of multiplication
3. Implement modular exponentiation and binary GCD → enable basic cryptographic primitives (RSA, Diffie-Hellman, etc.)

7. Conclusion

The project successfully demonstrates classic arbitrary-precision arithmetic algorithms and clearly illustrates the enormous performance difference between interpreted Python and compiled C++ code. The current implementation is a clean, correct, and well-tested prototype that provides an excellent foundation for further optimization and extension into a high-performance big-integer library.