■

1) Representation (the core design choice)

- Pick a limb/base representation. Two common choices:

A) Base = 2^32 (store as array of 32-bit unsigned integers) - Pros: natural fit for CPU word size, fast carry handling with native operations. - Cons: binary representation, prints need conversion to decimal string.

B) Base = 10^9 (or 10^7, 10^

8) (store as array of base-10 blocks)

- Pros: easier decimal conversion / printing; friendly for human-oriented tests. - Cons: a bit slower for low-level arithmetic compared with word-size base.

- Structure: an array/list of limbs (least-significant limb first) plus a sign flag (+1, 0, -

1). Keep limbs normalized so no leading zero limbs (except zero represented as empty or single zero limb).

- Important invariants: - No leading zero limbs. - Zero has sign = 0 (or sign = +1 and a separate zero check — pick one consistent rule). - Each limb is within [0, base-1].

- Memory layout: use a dynamic array (List/Array) that you can grow. Plan for in-place operations and immutable-style returns — decide early (mutable is faster but error-prone; immutable safer).

2) Public API (behavioral description — no code)

- Your class should expose these operations (behavior only):

- Constructor(s): - Create from 32-/64-bit integers, and from decimal string.

- Properties: - Sign property: negative/zero/positive. - BitLength/DecimalLength helpers (optional).

- Arithmetic: - Add(other) → returns new BigInt representing this + other. - Subtract(other) → returns this - other. - Multiply(other) → returns this * other. - DivideAndRemainder(divisor) → returns quotient and remainder (remainder sign as per convention: nonnegative and smaller than divisor magnitude). - Divide(divisor) and Remainder(divisor) separately optionally.

- Comparison: CompareTo(other), Equals. - Unary: Negate(), Abs().

- Utilities: - ToString() in decimal (and optional hex/bin). - Parse(string) to construct. - Increment/Decrement (optional). - LeftShift(bits)/RightShift(bits) (useful; implement as limb shifts + bit shifts). - ModPow(base, exponent, modulus) — modular exponentiation for speed. - Gcd(a, b) (optional but useful).

- Document: what happens on invalid input (e.g., divide by zero should throw).

3) Algorithms — conceptual descriptions

- Addition and Subtraction - Work limb-by-limb (least-significant limb first). - For addition: sum = a_i + b_i + carry. New carry = sum >= base. Store sum mod base. - For subtraction: compute a_i - b_i - borrow. If negative, borrow = 1 and add base to result limb.

- Sign handling: - If signs equal, do magnitude add and preserve sign. - If signs differ, do magnitude compare then subtract smaller magnitude from larger, sign = sign of larger magnitude.

- Complexity: O(n) where n = number of limbs of the longer operand.

- Multiplication - Start with schoolbook (grade-school) multiplication: - For each limb of A, multiply by each limb of B, accumulate into result limbs with carry. Complexity O(n*m) (~O(n^2) if lengths similar). - Implement using a wider intermediate type to hold product of two limbs + accumulated value (e.g., 64-bit intermediate when limbs are 32-bit).

- When to upgrade: - For inputs of a few thousand limbs or more, consider Karatsuba (divide-and-conquer) which is ~O(n^1.

5

8

5).

- For very large sizes (tens of thousands of digits/limbs), consider FFT/Schönhage–Strassen (convolutions). Those are complex and not needed for learning; note them as future improvements.

- Division (quotient + remainder) - Two conceptual approaches: - Long Division (schoolbook / classical algorithm): - Normalize divisor (ensure top limb >= base/2 if using base 2^k — leading limb nonzero) by scaling both numerator and divisor. - Iterate from the highest limb downward, estimate quotient limb by dividing top few limbs of remainder by top limb of divisor, refine (trial multiplication), subtract. This is the Knuth-style algorithm D. - Complexity roughly O(n*m) where n = dividend limbs, m = divisor limbs.

- Binary Long Division: - Use shifts and subtraction: repeatedly align divisor under dividend using left shifts and subtract where possible. Simpler conceptually but slower. - Edge case handling: - Division by zero -> error. - Remainder sign convention: prefer non-negative remainder with magnitude less than divisor magnitude.

Modular Arithmetic & Exponentiation

Modular multiplication: multiply then reduce (mod).

Modular exponentiation: use exponentiation by squaring (square-and-multiply) applied to big integers, using modular reduction at each step. Complexity O(log exponent * mult-cost).

Left/Right shifts (bitwise)

Implement shifts by entire limb (bits multiple of limb size) + intra-limb shifts with carry between limbs.

Useful for efficient multiplication by powers of two.

4) Normalization, canonical form, and invariants

After every arithmetic operation:

Remove leading zero limbs.

If the number is zero, canonicalize sign to zero or + depending on your chosen rule.

Ensure each limb within range.

Normalization prevents explosion of limbs and ensures equality checks are easy.

5) Complexity summary & when to switch algorithms

Add/Sub: O(n)

Schoolbook Multiply: $O(n^2)$

Karatsuba Multiply: ~O(n^1.

5

8

5) — switch when limbs >> a few hundred to thousands (benchmark to find threshold).

Long Division: O(n*m)

FFT-based multiplication: O(n log n) — only for very large sizes.

Advice: implement schoolbook first, measure performance on sizes you care about, then add Karatsuba if you need it.

6) Correctness & edge-case checklist

Handle these explicitly:

Zero operands.

Negative numbers (sign handling).

Different length operands.

Carry propagation across many limbs (e.g., adding 999... +

1).

Subtraction where result is zero.

Division where divisor > dividend (quotient = 0, remainder = dividend).

Division exactness (remainder

0).

Maximum limb values (overflow in intermediates — use wider type for intermediate product/carry).

Input parsing: invalid characters, leading signs, leading zeros, very long strings.

7) Unit tests and verification plan (examples — no code)

Create tests for each operation covering typical and edge cases. Example test vectors (decimal) you can use to validate behavior:

Addition/Subtraction:

0 + 0 = 0

1 + 999999999... (carry across many limbs)

123456789012345678901234567890 + 987654321098765432109876543210

(-

1

2

3) + 100 = -23

100 - 100 = 0

100 - 101 = -1

Multiplication:

0 * N = 0

1 * N = N

123456789 * 987654321 = 121932631112635269

Large same-digit multiplications to test carry propagation (e.g., repeating 9's × repeating 9's)

Division:

0 / n = 0 remainder 0

n / 1 = n remainder 0

10 / 3 => quotient 3, remainder 1

large number / slightly smaller large number => quotient 1, remainder difference

division where dividend < divisor => quotient 0, remainder dividend

division by zero => throws/exception

Modular & Exponentiation:

powmod(2, 10,

1

0

0

0) -> 24

powmod(1234567, 0, n) -> 1 (if n>

1)

powmod(0, 0, n) -> define convention (often

1) — document what you choose

Shift tests:

Left/Right shifts by multiples of limb size and by partial bits

Shifting zero must remain zero

Comparisons:

Equal magnitudes with different limb representations but normalized should be equal

Negative vs positive comparisons

Randomized testing:

Generate random large numbers (in decimal strings), use a trusted implementation (e.g., language BigInteger or an online big-int calculator) to compare results for add/sub/mul/div. This is extremely useful to catch subtle bugs.

Property testing:

Associativity/commutativity where applicable (add and multiply).

a - b + b == a

(a * b) / b (if b !=

0) yields a when exact.

8) Debugging tips & verification strategies

Start small: implement addition first, test extensively, then subtraction, then multiply, then division.

Visual checks: on failure, print limbs in hex or base-10-blocks. Compare intermediate limb arrays to expected.

Invariant asserts: add debug-only asserts to check normalization and limb ranges after operations.

Use reference BigInteger: .NET includes System.Numerics.BigInteger. Use it to verify results during development (but don’t ship reliance on it).

Step-by-step validation: for division, log quotient estimation, trial multiplication, correction steps to catch off-by-one estimates.

Unit test fuzzing: random test generations, cross-check with built-in BigInteger for thousands of cases.

Edge-case harness: test carry chains (e.g., all limbs max, then add

1), borrow chains (e.g., subtract where lots of zeros), and large multiplications.

9) Performance & implementation tips (without code)

Use the largest safe native intermediate type for limb products/carries. For 32-bit limb, use 64-bit intermediate to accumulate product + carry.

Avoid per-limb allocations. Re-use arrays or pre-allocate when possible.

Implement in-place operations when safe (be careful with aliasing).

Minimize bounds-check costs in inner loops (avoid high-level per-element checks in hot loops; in C# you can consider spans/unsafe after correctness is proven).

Choose an internal base that balances speed and decimal conversion costs. Many implementations use base 2^32 for speed.

When printing to decimal: use repeated division by 10^9 blocks or a base conversion algorithm rather than naive digit-by-digit extraction to be efficient.

For multiplication: optimize the inner loop for cache locality.

Profile: measure hot paths, and only optimize based on measurements.

1

0) Incremental implementation plan (step-by-step checklist you can follow)

Core data structure & normalization

Implement limb array, sign field, normalization routine.

Implement constructors from small ints and from simple decimal strings.

Implement ToString for debug (even if slow) to see results.

Addition

Add same-sign addition (magnitude add).

Add different-sign addition (use magnitude subtraction).

Unit tests for all edge cases.

Subtraction

Implement magnitude compare and subtraction with borrow.

Fix sign propagation rules.

Add tests: zero-result, negative results, borrow-chains.

Multiplication (schoolbook)

Implement schoolbook multiply using wide intermediate.

Test with small and medium inputs; cross-check with reference BigInteger.

Left/Right shift

Implement bit shifts (including limb shifts).

Useful for some division techniques and performance.

Division & remainder (long division)

Implement classical long division with quotient digit estimation and correction.

Carefully test with many edge cases and use debug logging for the estimation steps.

Utility methods

Implement Parse from string, ToString optimized.

Implement modular exponentiation and GCD.

Optimization pass

Benchmark typical sizes.

If needed, implement Karatsuba multiplication and switch based on operand size.

Polish

Add serialization, cloning, thread-safety notes.

Add XML/doc comments and API documentation.

1

1) Testing regimen (how many tests / types)

Unit tests: for each primitive op, include ~20 deterministic tests covering edge cases.

Randomized tests: run thousands of random tests of varying bit lengths, comparing to System.Numerics.BigInteger.

Property tests: verify algebraic properties (associativity, distributivity where expected).

Performance tests: measure time for operations on small, medium, large sizes (e.g., 32 bits, 1024 bits, 1e6 bits).

Regression tests: keep failing cases saved and re-run after changes.

1

2) Documentation & safety notes

Document the sign convention (especially for remainder).

Document thread-safety: by default, instance methods are not thread-safe unless immutable or synchronized.

Document expected input ranges and exceptions thrown (divide by zero, invalid parse).

1

3) References (free learning material)

Knuth, “The Art of Computer Programming”, Vol. 2 — algorithms for multi-precision arithmetic (if you want deeper theory; many university notes summarize the algorithms).

Online notes / PDFs about big integer arithmetic and Karatsuba / FFT multiplication (search for “big integer implementation algorithms” or “Knuth division algorithm D”).

.NET docs for System.Numerics.BigInteger to see behavior expectations (for testing).

1

4) Example debugging scenario (conceptual)

If a test fails for (A * B) where A and B are large:

Verify limb arrays for A and B are normalized.

Check the product of a single limb pair fits within the intermediate type.

Print intermediate accumulation for a specific pair of limbs.

Compare result with BigInteger to identify which digit/limb mismatch earliest — that points to the buggy region (either carry handling or incorrect index offsets).

If subtraction after multiplication (for division) fails, inspect quotient estimate and correction step.

1

5) Final checklist (what you should have when finished)

Normalized internal representation

Add/Sub operations (with sign handling, tested)

Multiply (schoolbook) and tests

Division + remainder (long division) and tests

Shift and compare operators

Parse/ToString (decimal) and tests

Modular exponentiation & GCD (optional)

Unit tests + randomized tests validated against reference BigInteger

Performance benchmarks and notes

Documentation for API behavior & invariants