Hands-On: Polynomial Multiplication

The Art of Computer Programming

26 November 2025

Objective. In this hands-on session, you will implement and experiment with different algorithms for multiplying univariate polynomials with integer coefficients. All programming tasks must be done in C++. As a *bonus*, feel free to later re-implement the same ideas in Python.

The C++ program must be split into three files:

`Polynomial.h` `Polynomial.cpp` `main.cpp`

Part A: Project Setup and Polynomial Representation

We will represent a univariate polynomial of degree $d \geqslant 1$

$$P(X) = \sum_{i=0}^{d} a_i X^i$$

with integer coefficients $a_i \in \mathbb{Z}$ as an *array of its coefficients*. If P is the zero polynomial, by convention we define its degree d to be -1 and consider its representation to be the empty array; otherwise, we assume $a_d \neq 0$. We will use `std::vector` in C++ to store this array. The vector provides coefficients in increasing degree order; all missing degrees must be represented by explicit zeros.

Q1. Create a new C++ project with three files:

- `Polynomial.h`
- `Polynomial.cpp`
- `main.cpp`

In `Polynomial.h`, declare a `class Polynomial` that stores a `std::vector<int>` in a private attribute. Do not forget the include guard. Declare a method `degree` to return the degree of the polynomial.

In `Polynomial.cpp`, implement the method `degree`.

Create a minimal `main.cpp` that includes `Polynomial.h` and for now only contains a `main` method creating a `Polynomial` with the default constructor of the class.

Test: Compile and run your program to check that the skeleton builds and runs correctly.

Part B: Constructors for Simple Polynomials

We now add basic constructors that will make it convenient to build polynomials.

2. Add the following constructors to the `Polynomial` class:
   - a constructor creating the zero polynomial;
   - a constructor taking a single integer coefficient c and creating the constant polynomial $P(X) = c$;
   - a constructor taking a `std::vector<int>` of coefficients `a` and constructing the polynomial $P(X) = a[0] + a[1]X + a[2]X^2 + \dots$ (ignoring trailing zeros).

   Update `Polynomial.h` accordingly, and implement these constructors in `Polynomial.cpp` unless their code is trivial.

   Test: In `main.cpp`, create at least the following polynomials:
   - a zero polynomial;

- a constant polynomial (e.g., 5);
- a polynomial from an array of coefficients, such as $1 + 2X + 3X^3$ and $4X^2 + 5X^4$.

For now you may just print simple confirmation messages to check that your code runs without crashing (e.g., of the form `std::cout << "Created polynomial of degree " << p.degree() << std::endl;`).

Part C: Basic Operations: Printing, Addition, Scalar Multiplication

We add some basic operations on polynomials.

3. Add a method `to_string` to `Polynomial` that returns a human-readable representation of the polynomial, for example:

   ```
   3X^4 - 2X^2 + 5
   ```

   You can use `std::to_string(i)` to get a string representation of an integer `i`.

   Test: In `main.cpp`, construct several polynomials and print `p.to_string()` using `std::cout`. Verify manually that the output matches the expected mathematical expression.

4. Add a method to add two polynomials; we will call this method `operator+` so that we can actually use it by writing `p+q` where `p` and `q` are two polynomials. It should be declared as follows:

   ```
   Polynomial operator+(const Polynomial& other) const;
   ```

   and return a new polynomial that contains the sum of the two input polynomials, properly combining coefficients of the same degree.

   Test: In `main.cpp`, test your method on various polynomials. Print the result using `to_string` and check correctness.

5. Add a method to multiply a polynomial on the right by an integer scalar, redefining `operator*`. As the first argument of the operator is the polynomial, we will use it by writing `p*42` where `p` is a polynomial.

   Test: In `main.cpp`, test:

   - multiplying by 0 (should give the zero polynomial),
   - multiplying by 1 (should give the same polynomial),
   - multiplying by a negative integer (check the signs).

6. We want to do the same but by multiplying a polynomial by an integer scalar *on the left*, e.g., writing `42*p`. This cannot be done with an `operator*` method, but it can be done with a `friend` function – check the lecture slides on object-oriented programming for an example. Implement this.

   Test: In `main.cpp`, run the same test as before, but now with left-multiplication.

Part D: Naïve Polynomial Multiplication

We start with the naïve (schoolbook) algorithm:

$$\left(\sum_{i=0}^{d_P} a_i X^i\right)\left(\sum_{j=0}^{d_Q} b_j X^j\right) = \sum_{k=0}^{d_P+d_Q}\left(\sum_{i+j=k} a_i b_j\right) X^k.$$

7. **(Naive multiplication: implementation)**
   In `Polynomial`, implement a method

   ```
   Polynomial multiply_naive(const Polynomial& other) const;
   ```

   using the straightforward double loop over monomials of the two polynomials.

   Test: In `main.cpp`, test `multiply_naive` on:

   - simple products: $(1 + X) \times (1 + X)$, $(1 + 2X + 3X^2) \times (1)$, etc.;
   - products involving zero polynomials;
   - products where both degrees are larger than 3 to check higher degrees.

   For each test, print the input polynomials and the result.

8. Let P and Q be polynomials of degrees d_P and d_Q, respectively. We assume either P or Q is non-zero, and we note $n = \max(d_P, d_Q)$.

   Express, using big-O notation, the asymptotic time complexity of `multiply_naive` in terms of n.

Part E: Divide-and-Conquer Multiplication

We now design a straightforward divide-and-conquer algorithm for multiplying two polynomials of "comparable" degrees (assume, for simplicity, that degrees are bounded by the same power of 2 and that you can pad with zeros).

Let n be a power of 2 and suppose P and Q have degree at most $n-1$. Write:

$$P(X) = A(X) + X^{n/2}B(X), \qquad Q(X) = C(X) + X^{n/2}D(X),$$

where A, B, C, D are polynomials of degree at most $n/2 - 1$.

Then:

$$P(X)Q(X) = AC + X^{n/2}(AD + BC) + X^n BD.$$

9. Implement a method

   ```
   Polynomial multiply_dc(const Polynomial& other) const;
   ```

   using a straightforward divide-and-conquer approach based on the above idea:

   a) if one of the polynomials is of degree $\leqslant 0$, multiply them directly;
   b) ensure both polynomials have the same "length" (pad with zero coefficients if needed);
   c) split each polynomial into its "low" part A (or C) and "high" part B (or D);
   d) recursively compute AC, AD, BC, BD;
   e) combine the results to obtain PQ.

   You may implement helper functions inside `Polynomial.cpp` for:

   - splitting a polynomial into low/high parts,
   - shifting a polynomial by multiplying by a power of X,
   - padding with zeros up to a given size.

   Test: In `main.cpp`, test `multiply_dc` on the same polynomial pairs as `multiply_naive` and check that the results are identical (e.g., by comparing `to_string()` outputs or by comparing coefficients). Also test on random polynomials of moderate degree.

10. Assume again $d_P = O(n)$ and $d_Q = O(n)$ and that your algorithm always works with sizes that are powers of 2. Let $T(n)$ denote the time needed to multiply two polynomials of degree at most $n-1$ using your divide-and-conquer procedure.

    a) Write a recurrence relation for $T(n)$.
    b) Use the Master Theorem to solve this recurrence and express the asymptotic complexity of this approach in big-O notation.
    c) Compare this complexity with that of the naive algorithm.

Part F: Karatsuba Multiplication

Karatsuba's algorithm is a clever improvement of the previous divide-and-conquer approach. Using the same notation as above, the straightforward method computes AC, AD, BC, BD (four multiplications). Karatsuba's idea is to compute $AD + BC$ more cleverly, using only *three* recursive multiplications instead of four.

11. Find a way to compute $AD + BC$ from A, B, C, D, AC, BD using a single extra multiplication.

12. Implement a method

    ```
    Polynomial multiply_karatsuba(const Polynomial& other) const;
    ```

    using a recursive Karatsuba algorithm:

    a) if one of the polynomials is of degree $\leqslant 0$, multiply them directly;
    b) otherwise, pad and split P and Q into A, B and C, D as before;

c) recursively compute the three products needed by Karatsuba;
d) combine them to get the result.

Test: In `main.cpp`, test `multiply_karatsuba`:

- on the same small examples as before, comparing to `multiply_naive`;
- on larger random polynomials, checking that the results agree (for example by comparing coefficients).

13. Let $T(n)$ denote the time to multiply two polynomials of degree at most $n-1$ using the Karatsuba algorithm, and assume again that we work with sizes that are powers of 2.

    a) Write a recurrence relation for $T(n)$ for Karatsuba's algorithm.

    b) Use the Master Theorem to obtain the asymptotic complexity of Karatsuba multiplication in big-O notation.

    c) Compare this complexity with the naive algorithm and with the straightforward divide-and-conquer algorithm.