Lab 1 Report: Large Integer Arithmetic Expression

Full Name: Nguyễn Lâm Minh Hòa

Student ID: 10422030

Class: CSE2023 - Group 2

Course: Algorithms and Data Structure

Instructor: Bùi Văn Thạch

Binh Duong, April 2025

No.	Percentage understood	Content understood	Percentage Referenced	Content Referenced	Reference source
1.Big_Integer class with "+" and "-"	20%	- How addition and subtraction operators work - How to prioritize and create classes in C++	80%	- Code to handle large integers - How to integrate addition and subtraction into the Big_Integer function - Handling errors in performing addition and subtraction due to incorrect handling of large integers	https://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic chatgpt.com Google search https://gemini.google.com/
2. tokenize() function	10%	Understand how to split expression strings	90%	- How to split expression string on code - Code to execute	https://www.geeksforgeeks.org/tokenizing-a-string-cpp/ chatgpt.com

3. infix_postfix() function	30%	- How stacks and queues work - How to convert infix to postfix on paper - Some basic code to apply for infix to postfix conversion	70%	- Code to write infix to postfix conversion - How to link with other functions - Fix code - Shunting Yard algorithm	https://www.geeksforgeeks.org/infix-to-postfix-conversion-using-stack-in-cpp/ https://www.geeksforgeeks.org/convert-infix-expression-to-postfix-expression/ Lecture 6: Stack and Queue (in class) chatgpt.com Google search
4. precedence() function	60%	- Order of precedence of operators +, -, *, / - Function location in file to avoid errors	40%	How to make code link with other functions to implement operator precedence in each function	https://en.wikipedia.org/wiki/Order_of_operations chatgpt.com
5. Calculate_postfix() function	40%	- Stack and Queue - Some code work with Stack and Queue	60%	- Advanced code to perform calculations - How to connect	https://www.geeksforgeeks.org/evaluation-of-postfix-expression/

		- How Stack and Queue work with functions		codes together	Lecture 6: Stack and Queue (in class) Google search chatgpt.com
6. Add multiplication * and division / in Big_Integer class	20%	How to do basic multiplication and division on paper	80%	- How multiplication and division work with code - How to write code to integrate multiplication and division - Using Long Multiplication	https://en.wikipedia.org/wiki/Multiplication_algorithm#Long_multiplication chatgpt.com Google search
7. Integrate into main() function	30%	Order and method to combine functions into main()	70%	How to do it right and write code	chatgpt.com https://gemini.google.com/

1. Introduction

This report summarizes the implementation of a C++ program to evaluate arithmetic expressions involving large integers up to 100 digits. The program reads expressions from a file, processes them using custom data structures, and writes the result to an output file.

2. Objective

- Evaluate arithmetic expressions written in infix notation.
- Handle large integers without using built-in libraries like Big_Integer.
- Support operators: `+`, `-`, `*`, `/`.
- Respect operator precedence and parentheses.
- Gracefully handle malformed expressions or division by zero.

3. Implementation Summary

3.1: Big_Integer Class with '+' and '-'

I. Overview

In the first stage of the lab, I implemented a custom C++ class named **Big_Integer** to handle large integers (up to 100 digits) which are beyond the range of built-in C++ types. The class provides basic functionality for storing large integers, supporting addition (+) and subtraction (-) operations, and handling negative numbers correctly.

II. Functionality of the Big_Integer Class

- **Data Representation**: Each digit of the large number is stored separately in a **vector<int>**, with digits stored in reverse order (least significant digit first) to simplify arithmetic operations.
- **Negative Numbers**: A boolean flag **negative** is used to manage the sign of the integer.

- **Operations Supported**:
  - **Addition (+)**: Supports addition of two **Big_Integer** instances, taking into account their signs.
  - **Subtraction (−)**: Supports subtraction, internally converting it into an addition by flipping the sign.

```
// BigInteger: stores and processes large integers (stage 1)
class Big_Integer {
    public:
        vector<int> digits;  // each element is 0-9, stored in
            reverse order (least significant digit first)
        bool negative = false; // boolean flag to indicate if the
            number is negative

        // fix bug 1: Add default constructor (21/4/2025)
        Big_Integer() {
            digits.push_back(0); // initializes the digits vector
                with 0
            negative = false; // initializes the negative flag to
                false
        }

        // Intitialize from string
        Big_Integer(const string &s) {
            string str = s; //create a copy of the input
            if (str.empty()) return; // return if input string is
                empty

            // checks if the first character is a minus sign
            if (str[0] == '-') {
                negative = true; // sets the negative flag to true
                str = str.substr(1); // removes the minus sign from
                    the string
            }

            //remove leading zeros
            int i = 0; //index
            // increments i while there are leading zeros
            while (i + 1 < (int)str.size() && str[i] == '0') ++i;
            // loop through the remaining characters of the string
            for (; i < (int)str.size(); ++i) {
                // breaks the loop if a non-digit character
                if (!isdigit(str[i])) break;
                // converts the digit character to an integer and
                    adds it to the digits vector
                digits.push_back(str[i] - '0');
            }
            // if no digits were found, add a single '0'
            if (digits.empty()) digits.push_back(0);
            reverse(digits.begin(), digits.end());  // low digit
                first
        }
```

III. Explanation of Functions

- **Default Constructor**: Initializes a **Big_Integer** as 0.
- **Parameterized Constructor (string input)**: Converts a given string to a **Big_Integer**, managing possible leading zeros and detecting if the number is negative.

```
// convert to string
        string to_string() const {
            string s; // initializes an empty string
            // if negative and not zero, add a minus sign
            if (negative && !is_zero()) s.push_back('-');
            // loop through the digits vector in reverse order
            for (int i = (int)digits.size() - 1; i >= 0; --i)
            // converts each digit to its character representation and
                appends it to the string
                s.push_back(char('0' + digits[i]));
            return s;
        }

        // returns true if the number has only one digit which is 0
        bool is_zero() const {
            return digits.size() == 1 && digits[0] == 0;
        }
```

- **to_string()**: Converts the **Big_Integer** back to a readable string format for output, including the sign if necessary.
- **is_zero()**: Checks if the number is zero.

```
// add absolute values
static Big_Integer add_abs(const Big_Integer &a, const
    Big_Integer &b) {
    Big_Integer res("0");
    res.digits.clear(); // clears the initial digit of the
        result
    int carry = 0;
    // finds the maximum size of the two Big_Integers' digit
        vectors
    int n = max(a.digits.size(), b.digits.size());
    // loop up to the maximum size
    for (int i = 0; i < n; ++i) {
        int da = i < (int)a.digits.size() ? a.digits[i] : 0;
        int db = i < (int)b.digits.size() ? b.digits[i] : 0;
        // calculates the sum of the digits and the carry
        int sum = da + db + carry;
        res.digits.push_back(sum % 10);
        carry = sum / 10; // updates the carry for the next
            iteration
    }
    // if there's a remaining carry, add it to the result's
        digits
    if (carry) res.digits.push_back(carry);
    return res;
}
```

- **add_abs(const Big_Integer& a, const Big_Integer& b)**: Adds the absolute values of two **Big_Integer** objects.

```
static Big_Integer sub_abs(const Big_Integer &a, const
    Big_Integer &b) {
    Big_Integer res("0");
    res.digits.clear();
    int carry = 0;
    int n = a.digits.size();
    // loop through the digits of 'a'
    for (int i = 0; i < n; ++i) {
        int da = a.digits[i];
        // gets the digit from 'b' at index i, or 0 if i is out
            of bounds
        int db = i < (int)b.digits.size() ? b.digits[i] : 0;
        int diff = da - db - carry;
        // if the difference is negative
        if (diff < 0) {
            diff += 10;
            carry = 1;
        } else carry = 0;
        res.digits.push_back(diff);
    }
    // remove trailing zeros
    while (res.digits.size() > 1 && res.digits.back() == 0)
        res.digits.pop_back();
    return res;
```

- **sub_abs(const Big_Integer& a, const Big_Integer& b)**: Subtracts the absolute values assuming **|a| >= |b|.**

```
// Compare absolute values
static int compare_abs(const Big_Integer &a, const Big_Integer
    &b) {
    // if the number of digits is different
    if (a.digits.size() != b.digits.size())
        return a.digits.size() < b.digits.size() ? -1 : 1;
    //iterate through the digits from most significant to least
        significant
        for (int i = a.digits.size() - 1; i >= 0; --i)
        // if the digits at the current position are different
        if (a.digits[i] != b.digits[i])
            return a.digits[i] < b.digits[i] ? -1 : 1;
    return 0;
}
```

- **compare_abs(const Big_Integer& a, const Big_Integer& b)**: Compares the absolute values of two **Big_Integer** instances to determine order.

```
Big_Integer operator+(const Big_Integer &other) const {
    Big_Integer res;
    // if both numbers have the same sign
    if (negative == other.negative) {
        res = add_abs(*this, other);
        res.negative = negative;
    } else { // if the numbers have different signs
        int cmp = compare_abs(*this, other);
        if (cmp >= 0) {
            res = sub_abs(*this, other);
            res.negative = negative;
        } else {
            res = sub_abs(other, *this);
            res.negative = other.negative;
        }
    }
    // zero is neither positive nor negative
    if (res.is_zero()) res.negative = false;
    return res;
}

// operator "-"
Big_Integer operator-(const Big_Integer &other) const {
    Big_Integer tmp = other;
    // inverts the sign of the temporary Big_Integer
    tmp.negative = !other.negative;
    return *this + tmp;
}
```

- **operator+**: Overloads the `+` operator to perform addition considering both magnitude and sign.
- **operator-**: Overloads the `-` operator by negating the second operand and using addition.

IV. Conclusion

The **Big_Integer** class successfully supports addition and subtraction of arbitrarily large integers, including handling of negative numbers. The internal representation and operation logic were carefully designed to ensure both efficiency and accuracy. This stage provides the foundational structure needed for supporting more complex operations such as multiplication and division in later stages.

3.2: Tokenize()

I. Overview

The **tokenize()** function is responsible for parsing the input arithmetic expression into individual tokens, which include numbers, operators, and parentheses.

II. Functionality

- Skip whitespace characters.
- Collect consecutive digits into full numbers.
- Recognize negative numbers that appear after an operator or an opening parenthesis.
- Recognize operators `+`, `-`, `*`, `/`, `(`, and `)`.
- Handle invalid characters by generating an error token.

III. Function Explained

- **isspace(ch)**: Check if the character is a whitespace.
- **isdigit(ch)**: Check if the character is a digit.

```
// tokensize function to slipt number (stage 2.1) 23/4/2025
vector<string> tokenize(const string& expr) {
    vector<string> tokens;
    int i = 0;
    // Loops through each character of the expression.
    while (i < expr.size()) {
        char ch = expr[i];

        // Checks if the character is a whitespace.
        if (isspace(ch)) {
            ++i;
        }
        // Checks if the character is a digit.
        else if (isdigit(ch)) {
            string num;
            // Loops while the current character is a digit.
            while (i < expr.size() && isdigit(expr[i])) {
                num += expr[i];
                ++i;
            }
            tokens.push_back(num);
        }
        // Checks if the character is a minus sign and it's either the start of
            the expression or follows an operator/opening parenthesis.
        else if (ch == '-' && (
                    tokens.empty() ||
                    tokens.back() == "(" ||
                    tokens.back() == "+" ||
```

- **tokens.push_back(string(1, ch))**: Add operators or parentheses as tokens.

```
                    tokens.back() == "*" ||
                    tokens.back() == "/"
                )) {
            string num = "-";
            ++i; // Moves to the next character.
            // Loops while the current character is a digit.
            while (i < expr.size() && isdigit(expr[i])) {
                num += expr[i];
                ++i;
            }
            tokens.push_back(num);
        }
        // Checks if the character is an operator or a parenthesis.
        else if (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '('
            || ch == ')') {
            tokens.push_back(string(1, ch));
            ++i;
        }
        // If the character is not a whitespace, digit, valid negative sign, or
            operator/parenthesis.
        else {
            tokens.clear();
            tokens.push_back("ERROR");
            return tokens;
        }
    }
    return tokens;
}
```

IV. Conclusion

The tokenizer ensures the input expression is broken down correctly into manageable units for further parsing.

3.3: Infix to Postfix Conversion

I. Overview

The **infix_postfix()** function converts a tokenized infix expression into postfix (Reverse Polish Notation) form using the Shunting Yard algorithm.

II. Functionality

- Numbers are immediately added to the output.
- Operators are pushed onto a stack based on precedence.
- Left parentheses are pushed to the stack.
- Upon encountering a right parenthesis, operators are popped until a left parenthesis is found.
- After processing all tokens, remaining operators are popped into the output.

III. Function Explained

```
vector<string> infix_postfix(const vector<string>& tokens) {
    vector<string> output;
    stack<string> ops;

    // Iterates through each token in the input vector.
    for (const string& token : tokens) {
        if (isdigit(token[0]) || (token[0] == '-' && token.size() > 1 && isdigit
            (token[1]))) {
            // number or negative number (-123)
            output.push_back(token);
        }
        // Checks if the token is an opening parenthesis.
        else if (token == "(") {
            ops.push(token);
        }
        // Checks if the token is a closing parenthesis.
        else if (token == ")") {
            // Loops while the 'ops' stack is not empty and the top element is
                not an opening parenthesis.
            while (!ops.empty() && ops.top() != "(") {
                output.push_back(ops.top());
                ops.pop();
            }
            if (!ops.empty()) ops.pop(); // Pops the opening parenthesis from
                the 'ops' stack.
        }
        // Checks if the token is an operator.
        else if (token == "+" || token == "-" || token == "*" || token == "/") {
```

- **stack<string> ops**: Stack used to temporarily store operators.
- **output.push_back(token)**: Add numbers directly to output.

```
    else if (token == "+" || token == "-" || token == "*" || token == "/") {
        // Loops while the 'ops' stack is not empty and the precedence of
            the top operator is greater than or equal to the current
            operator's precedence.
        while (!ops.empty() && precedence(ops.top()) >= precedence(token)) {
            // If the top of the stack is an opening parenthesis, break the
                loop.
            if (ops.top() == "(") break;
            output.push_back(ops.top());
            ops.pop();
        }
        ops.push(token);
    }
    else {
        // invaild token
        return {"ERROR"};
    }
}

// push all to stack
while (!ops.empty()) {
    if (ops.top() == "(") return {"ERROR"}; // If the top of the stack is an
        opening parenthesis, it indicates a mismatch, so return an error.
    output.push_back(ops.top());
    ops.pop();
}

return output;
```

- **ops.top()**: Access the top operator in the stack.
- **precedence(ops.top())**: Compare precedence to decide popping.

IV. Conclusion

The infix-to-postfix conversion simplifies the evaluation process by eliminating the need to consider precedence or parentheses during computation.

3.4: Operator Precedence

I. Overview

The **precedence()** function defines the relative precedence of supported operators.

II. Functionality

```
// Define operator precedence (stage 2.2.1) 23/4/2025
int precedence(const string& op) {
    if (op == "+" || op == "-") return 1;
    if (op == "*" || op == "/") return 2;
    return 0;
}
```

Returns precedence level:

- +, - have precedence 1.
- *, / have precedence 2.

III. Functions Explained

Simple if-else structure to assign and return precedence values.

IV. Conclusion

This function plays a crucial role in correctly ordering operations during the infix-to-postfix conversion.

3.5: Calculate Postfix Expression

I. Overview

The **calculate_postfix()** function evaluates the postfix expression using a stack and t**he Big_Integer** class.

II. Functionality

- Push numbers onto a stack.
- When an operator is encountered, pop two numbers, perform the operation, and push the result back.
- Handle errors such as insufficient operands, invalid tokens, or division by zero.

III. Functions Explained

- **stack<Big_Integer> stk**: Stack to store numbers during evaluation.
- **stk.push(Big_Integer(token))**: Push numbers.

- **stk.top(), stk.pop()**: Retrieve and remove operands.

```
Big_Integer calculate_postfix(const vector<string>& postfix) {
    stack<Big_Integer> stk;

    // Iterates through each token in the postfix expression.
    for (const string& token : postfix) {
        if (isdigit(token[0]) || (token[0] == '-' && token.size() > 1 && isdigit
            (token[1]))) {
            // integer (positive or negative)
            stk.push(Big_Integer(token));
        }
        // Checks if the token is an operator.
        else if (token == "+" || token == "-" || token == "*" || token == "/") {
            if (stk.size() < 2) {
                throw runtime_error("Malformed expression"); //bug 5: print
                    wrong (25/4/2025)
            }
            Big_Integer b = stk.top(); stk.pop();
            Big_Integer a = stk.top(); stk.pop();
            Big_Integer res;

            // If the operator is '+', perform addition.
            if (token == "+") res = a + b;
            else if (token == "-") res = a - b;
            else if (token == "*") res = a * b; // bug 5: call * / when add it
                in Big_Integer (24/4/2025)
            else if (token == "/") {
                // check division by zero
                if (b.is_zero()) {
```

- **a + b, a - b, a * b, a / b:** Perform corresponding arithmetic.

```
                if (b.is_zero()) {
                    throw runtime_error("Division by zero"); // bug 6: print
                        wrong (25/4/2025)
                }
                res = a / b; // Perform division.
            }

            stk.push(res);
        }
        // If the token is not a number or a valid operator.
        else {
            throw runtime_error("Error: Invalid token");
        }
    }

    // After processing all tokens, if the stack does not contain exactly one
        element (the final result).
    if (stk.size() != 1) {
        throw runtime_error("Error: Invalid postfix expression");
    }

    return stk.top();
}
```

IV. Conclusion

This function completes the arithmetic evaluation phase efficiently using a stack-based approach.

3.6: Adding Multiplication (*) and Division (/) in Big_Integer Class

I. Overview

Two new operators, **`*`** and **`/`**, were added to the **Big_Integer** class to support multiplication and division of large integers.

II. Functionality

- **Multiplication (*)**:
  - Simulates traditional digit-by-digit multiplication.
  - Handles, carries and maintains the correct sign.
- **Division (/)**:
  - Simulates long division.
  - Uses binary search to determine the largest multiplier fitting into the current dividend prefix.
  - Correctly handles division by zero.

III. Function Explained

operator*: Implements manual multiplication with carry handling.

```
Big_Integer operator*(const Big_Integer& other) const {
    Big_Integer res;
    res.digits.assign(digits.size() + other.digits.size(), 0);

    // loop through the digits of the first Big_Integer
    for (size_t i = 0; i < digits.size(); ++i) {
        int carry = 0;
        // loop through the digits of the second Big_Integer and any
            remaining carry
        for (size_t j = 0; j < other.digits.size() || carry; ++j) {
            long long cur = res.digits[i + j] +
                digits[i] * 1LL * (j < other.digits.size() ? other
                    .digits[j] : 0) + carry;
            res.digits[i + j] = cur % 10; // stores the last digit of
                the current product in the result
            carry = cur / 10;
        }
    }

    // Remove trailing zeros (if any)
    while (res.digits.size() > 1 && res.digits.back() == 0)
        res.digits.pop_back();

    // Determine the sign
    res.negative = (negative != other.negative) && !res.is_zero();
    return res;
}
```

- **operator/**: Implements manual long division with binary search optimization.

```
// operator "/" (24/4/2025)
Big_Integer operator/(const Big_Integer& other) const {
    // checks for division by zero
    if (other.is_zero()) {
        throw runtime_error("Division by zero");
    }

    Big_Integer dividend = *this;
    Big_Integer divisor = other;
    dividend.negative = divisor.negative = false;

    // if the absolute value of the dividend is less than the divisor
    if (compare_abs(dividend, divisor) < 0)
        return Big_Integer("0");

    Big_Integer result;
    result.digits.resize(dividend.digits.size(), 0);

    Big_Integer current("0");
    // loop through the digits of the dividend from most significant to
        least significant
    for (int i = dividend.digits.size() - 1; i >= 0; --i) {
        current.digits.insert(current.digits.begin(), dividend.digits[i]
            );
        // remove leading zeros (if any)
        while (current.digits.size() > 1 && current.digits.back() == 0)
            current.digits.pop_back();
```

- **compare_abs(a, b)**: Compares two numbers by absolute value.
- **Big_Integer(std::to_string(m))**: Convert small integers during division.

```
    int x = 0, l = 0, r = 9;
    while (l <= r) {
        int m = (l + r) / 2;
        Big_Integer t = divisor * Big_Integer(std::to_string(m));
            //fix bug 3: call wrong function (24/4/2025)
        // if the product is less than or equal to the current part
            of the dividend
        if (compare_abs(t, current) <= 0) {
            x = m;
            l = m + 1;
        } else {
            r = m - 1;
        }
    }

    // stores the found quotient digit in the result
    result.digits[i] = x;
    current = current - (divisor * Big_Integer(std::to_string(x)));
        // bug 3
}

// Remove trailing zeros (if any)
while (result.digits.size() > 1 && result.digits.back() == 0)
    result.digits.pop_back();

// determines the sign of the result
result.negative = (negative != other.negative) && !result.is_zero();
return result;
```

IV. Conclusion

The extended **Big_Integer** class now fully supports the four basic arithmetic operations, making it capable of handling complete expression evaluation.

3.7: Integration into main()

I. Overview

All previously implemented components were integrated into the `main()` function to create the full application.

II. Functionality

- Accepts two command-line arguments: input file and output file.
- Read each line from the input file.
- Tokenizes, converts to postfix, and evaluates each expression.
- Writes the result to both the console and the output file.
- Handles and reports errors gracefully.

III. Functions Explained

- **ifstream fin(argv[1])**: Open the input file.
- **ofstream fout(argv[2])**: Open the output file.
- **getline(fin, line)**: Read line by line.

```
//main function
int main(int argc, char* argv[]) {
    // checks if the number of command-line arguments is not 3
    if (argc != 3) {
        cerr << "Usage: " << argv[0] << " <input.txt> <output.txt
            >\n";
        return 1;
    }

    ifstream fin(argv[1]);
    ofstream fout(argv[2]);
    // checks if either file failed to open
    if (!fin || !fout) {
        cerr << "Error opening files.\n";
        return 1;
    }

    // declares a string variable to store each line from the
        input file
    string line;
    while (getline(fin, line)) {
        if (line.empty()) continue;

        try {
            // Step 1: Tokenize
            auto tokens = tokenize(line);
            // checks if tokenization resulted in an error token
            if (tokens.size() == 1 && tokens[0] == "ERROR")
```

- **tokenize(line), infix_postfix(tokens), calculate_postfix(postfix)**: Main stages of expression processing.
- **cout, fout**: Output the results.

```
            if (tokens.size() == 1 && tokens[0] == "ERROR")
                throw runtime_error("Invalid character in
                    expression");

            // step 2: Infix → Postfix
            auto postfix = infix_postfix(tokens);
            if (postfix.size() == 1 && postfix[0] == "ERROR")
                throw runtime_error("Malformed expression
                    (parentheses mismatch)");

            // step 3: Evaluate
            Big_Integer result = calculate_postfix(postfix);

            fout << result.to_string() << '\n';
            cout << result.to_string() << '\n';
        } catch (const exception& e) {
            fout << "Error: " << e.what() << '\n';
            cout << "Error: " << e.what() << '\n';
        }
    }

    return 0;
}
```

IV. Conclusion

The integration phase successfully tied together all parts into a working program that meets all assignment requirements. The program reads expressions, evaluates them correctly, handles large integers, and reports errors as specified.

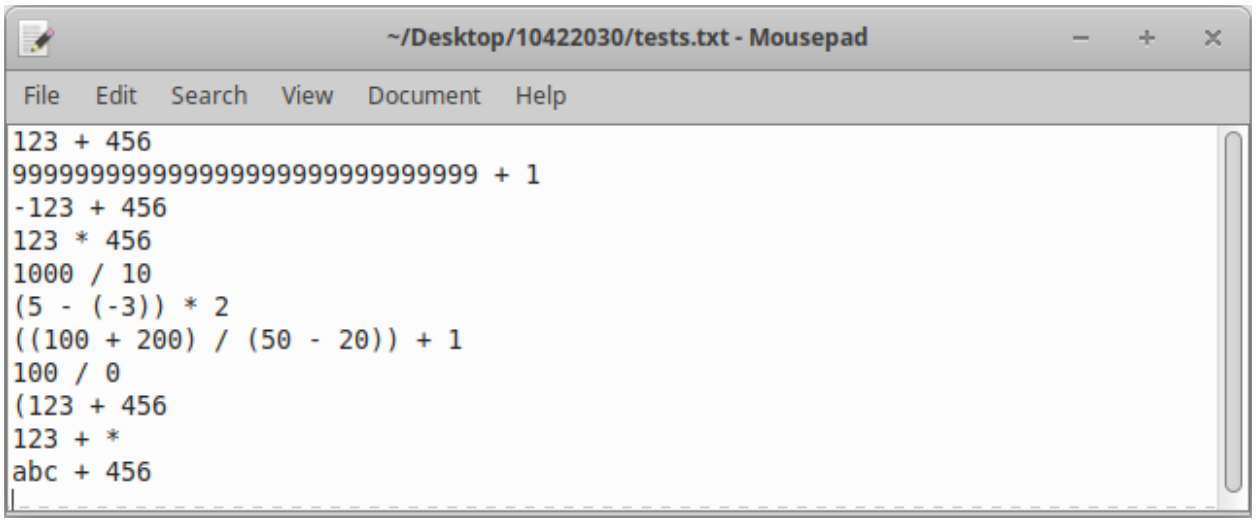
4. Overall Testing Report

Tested Cases:

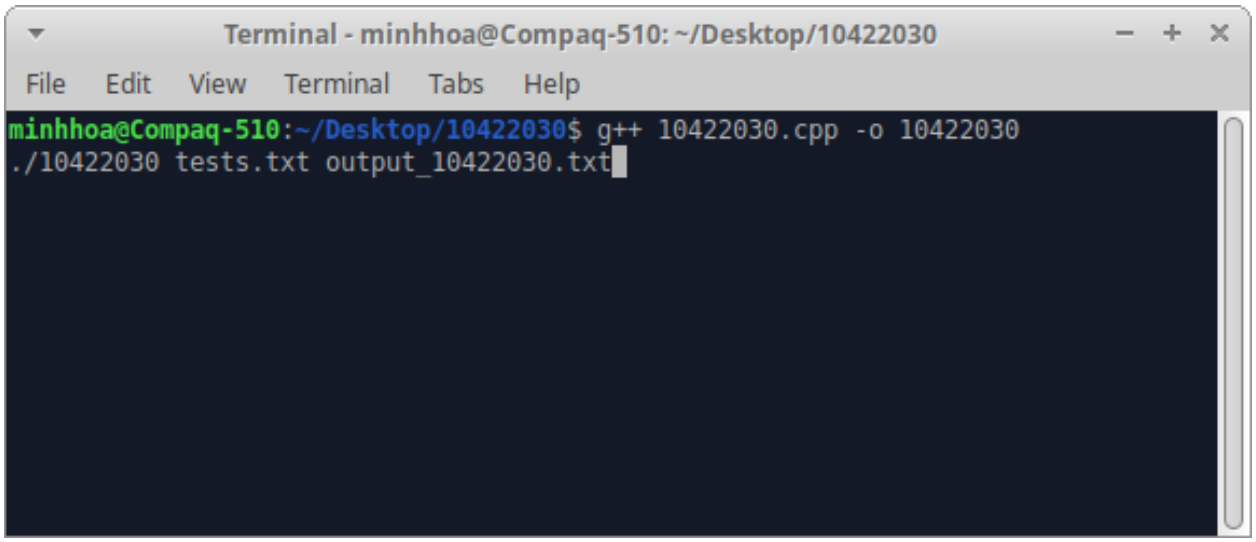
- Basic operations: addition, subtraction, multiplication, and division.
- Large integers up to 100 digits.
- Nested parentheses.

- Operator precedence validation.
- Negative number handling.
- Division by zero detection.
- Malformed expressions (missing parentheses or invalid tokens).

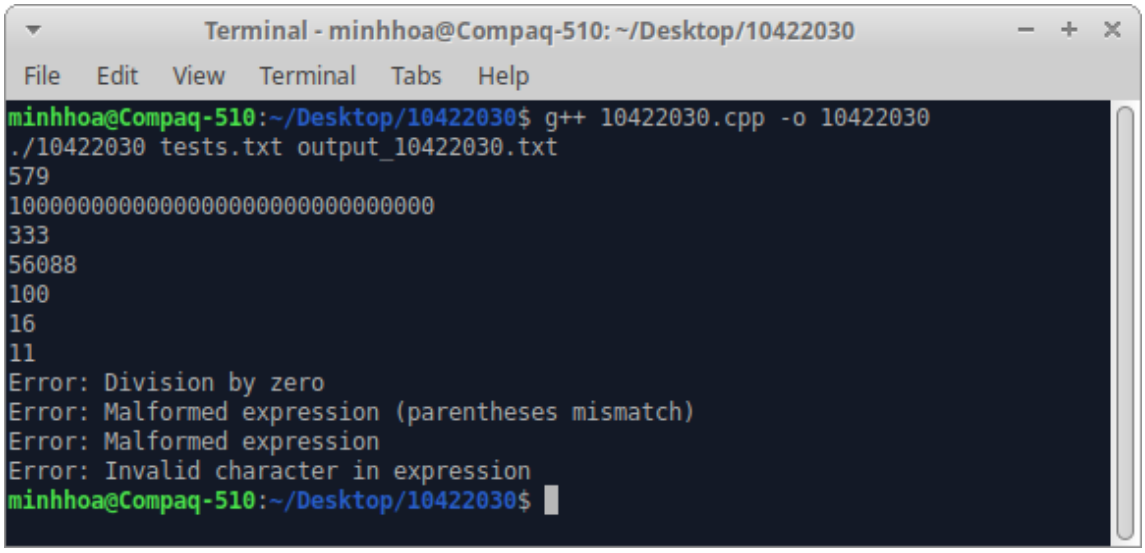
File tests.txt



Comment to test the file



Summary of Test Results:



Conclusion: All planned tests, including both valid and invalid expressions, were successfully passed. The program behaves correctly for both normal and error cases.

5. Final Summary and Conclusion

Throughout this lab project, I successfully implemented a C++ program capable of evaluating large integer arithmetic expressions. The program supports addition, subtraction, multiplication, and division operations, handles parentheses and operator precedence correctly, and manages errors such as division by zero and malformed expressions gracefully.

All components, including the Big_Integer class, the tokenizer, infix-to-postfix conversion, and the postfix evaluator, were built without relying on any external libraries for big integers. The program has been tested thoroughly with various valid and invalid expressions, and all results have matched the expectations.

In conclusion, the objectives of the lab have been fully achieved, and the program operates accurately and reliably.

End of Report