Model:TheB.AI Free

Problem Statement: Create a big integer library using C++ (similar to the one available in Java). The library should provide functionalities to store arbitrarily large integers and perform basic math operations.

Operations:
1. Addition(+), subtraction(-), multiplication(x, lowercase “X”), division(/)
E.g.
Input: 32789123+99893271223x9203232392-4874223
Output: 919340989462382970316
Input: 3423542525+6773442x5345345-213213197786/45647
Output: 36209803199102
2. Exponentiation
- Base will be a big int and exponent will be < 263
3. GCD of two numbers
4. Factorial

Constraints: For all the operations, input will be such that the number of digits in output won’t exceed 3000 digits.

Input Format: First line will contain an integer value which denotes the type of operation. The integer value and operation mapping is as follows:
1. Addition, Subtraction, Multiplication & Division
2. Exponentiation
3. GCD
4. Factorial

The following line will contain input according to the type of operation. For 1st and 4th type of operation, there will be one string and for the 2nd & 3rd type of operation, there will be 2 space separated Strings.

Evaluation Parameters: Accuracy of operations and performance

Sample Cases:
● Sample input:
1
1+2x6+13/5-2
Sample output:
13

● Sample input:
2
2 10
Sample output:
1024

● Sample input:
3
9 15
Sample output: 3

● Sample input:
4
12
Sample output: 362880

Note :
1. Negative numbers won’t be present in the intermediate or final output (i.e. No need to consider cases like 2-3).
2. There are NO brackets in the input.
3. Perform Integer division operation between two big integers, disregarding the remainder.
4. Addition, Subtraction, Multiplication and Division follows the same precedence and associativity rules as in Java/cpp.
5. Ignore Division by zero, gcd(0, x), gcd(x, 0).
6. C++ STL is NOT allowed (vectors & stacks are NOT allowed as well, design your own if required).
7. You are NOT allowed to use the regex library.
8. string, to_string and string manipulation methods are allowed.
9. Design your main function according to sample input/output given.

2023-08-13 16:03:29 Copy

To implement a big integer library in C++, we can follow the following approach:

1. Create a class called "BigInteger" that will store and manipulate the big integers.
2. The class should have a private data member of type string to store the digits of the big integer.
3. The class should have private member functions to perform basic math operations like addition, subtraction, multiplication, and division on big integers.
4. The class should also have public member functions to perform exponentiation, find the GCD of two big integers, and calculate the factorial of a big integer.
5. The class should also have public member functions to input and output big integers.
6. Implement the basic math operations by converting the big integer's digits string to individual digits and performing the operations digit by digit.
7. For multiplication and division, you can use the standard long multiplication and long division algorithms, respectively.
8. Implement the exponentiation function using repeated multiplication.
9. To calculate the GCD, you can use the Euclidean algorithm.
10. To calculate the factorial, use a loop to multiply the big integer by the numbers from 1 to the given input.

Here is an example implementation of the BigInteger class:

cpp Copy code

```cpp
#include <iostream>
#include <string>
using namespace std;

class BigInteger {
private:
    string digits;

    int charToInt(char c) {
        return c - '0';
    }

    char intToChar(int num) {
        return num + '0';
    }

    void removeLeadingZeroes() {
        int i = 0;
        while (i < digits.length() && digits[i] == '0') {
            i++;
        }
        digits = digits.substr(i);
    }

    void addLeadingZeroes(int numZeroes) {
        string zeroes(numZeroes, '0');
        digits = zeroes + digits;
    }

public:
    BigInteger() {
        digits = "0";
    }

    BigInteger(string str) {
        digits = str;
    }

    BigInteger operator+(const BigInteger& other) {
        string result;
        int carry = 0;
        int i = digits.length() - 1;
        int j = other.digits.length() - 1;
```

```
        while (i >= 0 || j >= 0 || carry > 0) {
            int digit1 = (i >= 0) ? charToInt(digits[i--]) : 0;
            int digit2 = (j >= 0) ? charToInt(other.digits[j--]) : 0;
            int sum = digit1 + digit2 + carry;
            carry = sum / 10;
            result = intToChar(sum % 10) + result;
        }
        return BigInteger(result);
    }

    BigInteger operator-(const BigInteger& other) {
        string result;
        int borrow = 0;
        int i = digits.length() - 1;
        int j = other.digits.length() - 1;

        while (i >= 0) {
            int digit1 = charToInt(digits[i--]);
            int digit2 = (j >= 0) ? charToInt(other.digits[j--]) : 0;
            int diff = digit1 - digit2 - borrow;
            if (diff < 0) {
                borrow = 1;
                diff += 10;
            } else {
                borrow = 0;
            }
            result = intToChar(diff) + result;
        }
        BigInteger resultBigInt(result);
        resultBigInt.removeLeadingZeroes();
        return resultBigInt;
    }

    BigInteger operator*(const BigInteger& other) {
        string result(digits.length() + other.digits.length(), '0');

        for (int i = digits.length() - 1; i >= 0; i--) {
            int carry = 0;
            int digit1 = charToInt(digits[i]);

            for (int j = other.digits.length() - 1; j >= 0 || carry > 0; j--) {
                int digit2 = (j >= 0) ? charToInt(other.digits[j]) : 0;
                int prod = digit1 * digit2 + carry;
                carry = prod / 10;
                int index = i + j + 1;
                int sum = charToInt(result[index]) + prod % 10;
                result[index] = intToChar(sum % 10);
                carry += sum / 10;
            }
        }
        BigInteger resultBigInt(result);
        resultBigInt.removeLeadingZeroes();
        return resultBigInt;
    }

    BigInteger operator/(const BigInteger& other) {
        BigInteger dividend(digits);
        BigInteger divisor(other);
        BigInteger quotient;
        BigInteger zero("0");

        if (divisor == zero) {
            throw "Division by zero";
        }

        while (dividend >= divisor) {
            int numZeroes = dividend.digits.length() - divisor.digits.length();
            divisor.addLeadingZeroes(numZeroes);
            int count = 0;
```

```
            while (dividend >= divisor) {
                dividend = dividend - divisor;
                count++;
            }

            quotient.digits += intToChar(count);
            quotient.removeLeadingZeroes();
            divisor.digits = divisor.digits.substr(1);
        }
        return quotient;
    }

    BigInteger pow(int exp) {
        BigInteger base(digits);
        BigInteger result("1");

        while (exp > 0) {
            if (exp & 1) {
                result = result * base;
            }
            base = base * base;
            exp >>= 1;
        }
        return result;
    }

    BigInteger gcd(const BigInteger& other) {
        BigInteger a(digits);
        BigInteger b(other);

        while (b != BigInteger()) {
            BigInteger temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }

    BigInteger factorial() {
        BigInteger result("1");

        for (int i = 1; i <= stoi(digits); i++) {
            BigInteger num(to_string(i));
            result = result * num;
        }
        return result;
    }

    friend istream& operator>>(istream& input, BigInteger& bigInt) {
        input >> bigInt.digits;
        return input;
    }

    friend ostream& operator<<(ostream& output, const BigInteger& bigInt) {
        output << bigInt.digits;
        return output;
    }

    bool operator==(const BigInteger& other) {
        return digits == other.digits;
    }

    bool operator>(const BigInteger& other) {
        if (digits.length() != other.digits.length()) {
            return digits.length() > other.digits.length();
        } else {
            return digits > other.digits;
        }
    }
```

```
    bool operator<(const BigInteger& other) {
        if (digits.length() != other.digits.length()) {
            return digits.length() < other.digits.length();
        } else {
            return digits < other.digits;
        }
    }

    bool operator>=(const BigInteger& other) {
        return *this > other || *this == other;
    }

    bool operator<=(const BigInteger& other) {
        return *this < other || *this == other;
    }
};

int main() {
    int operation;
    cin >> operation;

    if (operation == 1) {
        string input;
        cin >> input;
        BigInteger result;

        for (int i = 0; i < input.length(); i++) {
            if (input[i] == '+') {
                result = result + BigInteger(input.substr(i + 1));
                break;
            } else if (input[i] == '-') {
                result = result - BigInteger(input.substr(i + 1));
                break;
            } else if (input[i] == 'x' || input[i] == 'X') {
                result = result * BigInteger(input.substr(i + 1));
                break;
            } else if (input[i] == '/') {
                result = result / BigInteger(input.substr(i + 1));
                break;
            }
        }
        cout << result << endl;

    } else if (operation == 2) {
        int base, exp;
        cin >> base >> exp;
        BigInteger result(to_string(base));
        result = result.pow(exp);
        cout << result << endl;

    } else if (operation == 3) {
        int num1, num2;
        cin >> num1 >> num2;
        BigInteger result1(to_string(num1));
        BigInteger result2(to_string(num2));
        result1 = result1.gcd(result2);
        cout << result1 << endl;

    } else if (operation == 4) {
        int num;
        cin >> num;
        BigInteger result(to_string(num));
        result = result.factorial();
        cout << result << endl;
    }

    return 0;
}
```

This implementation defines the BigInteger class with all the required operations like addition, subtraction, multiplication, division, exponentiation, GCD, and Factorial. It also overloads the input and output operators for easy input and output of big integers.

The main function reads the operation type from input and performs the corresponding operation based on the given input. Finally, it prints the result.

2023-08-13 16:09:53 Copy Plaintext