Big Unsigned Interger Project

Introduction

The objective of this project is to gain a comprehensive understanding of the fundamental principles of object-oriented programming in both C++ and Python, with a particular emphasis on optimizing algorithms and achieving expert-level proficiency in utilizing libraries for multiprocessing.

Reporting

The report is divided into several sections:

- Problems and Choices of Solving
- Code Explanations
- Results
- Conclusion
- Future Directions

Author

Tran Le Minh

Date

4 April 2022

Table

Python and C++ Library Choices

Python Library:

- Numpy: This is a crucial library as one of the Python versions relies on Numpy extensively for its numerical computing capabilities.
- Numba: This library is designed for GPU multiprocessing. Even this project does not include the code using Numba because it's failed but I will talk about it a little bit.
- Multiprocessing: As the name suggests, this library enables efficient multiprocessing on CPU, allowing us to take advantage of the full potential of our hardware.

C++ Library:

- String, Vector, Array, Algorithm, and more: These widely-used libraries are integral components of any C++ project, providing essential functionality and flexibility.
- Pthread: Similar to Python's multiprocessing library, Pthread enables efficient CPU multiprocessing, allowing for optimal utilization of system resources.

Problems and Choices of Solving

Problems:

The objective of this task is to construct a class called UintN that can effectively represent unsigned integers of extremely large magnitude. This requires defining arithmetic and comparison operations that are optimized to handle large integers with speed and efficiency. In addition, a method must be implemented to determine the primality of a given UintN. Optimize the data structure to prevent the program from slowing down with large numbers.

	C++	**Python**
Integer format	String	String and Numpy array
Arithmetic and comparision operations definition	Elementary method	
Input method definition	Constructors that allow input from a string, int, unsigned long long, and other input types	
Determining primality	Miller-Rabins Test	

Additional methods include utilizing binary form for computation and comparison, as well as implementing vector<int> or vector<char> in C++. Nevertheless, the outcomes produced were unsatisfactory, leading to the abandonment of these approaches.

Regarding the method for testing primality, it is planned to incorporate AKS and other enhanced techniques based on existing algorithms in the near future, in addition to the Miller-Rabin method, with the aim of achieving superior outcomes.

Choices of Solving:

After a brief summary of the problem and solving method, in this section I will discuss about why I choose them.

- Interger format

  The string library is a widely used built-in library in both Python and C++, making looping through strings a highly efficient process, and optimizing memory usage by using a string instead of an int or char array is much better (Python Array version will show this result). From this, it can be inferred that using string conversion will yield much better results than using arrays or binary conversion.

- Arithmetic and comparision operations definition

  Converting integers to binary form can lead to numerous complications, and converting extremely large binary numbers can be a time-intensive process without the use of pre-existing libraries. Additionally, pre-existing libraries are often unable to handle binary numbers larger than 2^{64}. As a result, the binary conversion method is not utilized.

- Determining primality

  The main reason for abstaining from the brute force method (i.e. looping until the square root of the number) is its excessive time consumption. Consequently, we are compelled to rely on primality test techniques. Although these methods may still exhibit some degree of error, the magnitude of such errors is negligible, thereby rendering the resulting solutions still acceptable. For instance, the Miller-Rabin method that i implement entails an error rate $< \frac{1}{4^5} \approx 0.097\%$.

To sum up, the utilization of multiprocessing in Primality testing has significantly improved the speed of generating results. Nevertheless, when using an array to store numbers, the speed is still considerably slow, even with the implementation of vector<long long> to store each element with nine digits per element.

4

Code Explanations

This section covers explanations of the algorithms employed. Given the similarities in implementation between C++ and Python, with C++ possessing some specific enhancements, the emphasis will mainly be placed on elucidating the C++ code, accompanied by a brief exposition of the Python code.

Constructor and Helper Functions:

```
//Constructor
UintN(string a);
UintN(const UintN& a) : BigInt(a.BigInt) {}
// UintN(unsigned long long n)
{
    do
    {
        BigInt.push_back(n % 10);
        n /= 10;
    } while (n);
}
UintN() { BigInt.push_back(0); }

//Helper
friend int size(const UintN& a) { return a.BigInt.size(); }
friend bool Null(const UintN& a)
{
    if (a.BigInt.size() == 1 && a.BigInt[0] == 0)
        return true;
    return false;
}

friend long long sum_digits(UintN a)
{
    long long sum = 0;
    for (int i = a.BigInt.size() - 1; i >= 0; --i)
        sum += (short)a.BigInt[i];
    return sum;
}
friend bool sign_dividable_7(UintN a);
friend bool sign_dividable_11(UintN a);
```

There are a total of four constructors available, each representing different ways of initializing variables. For the constructors that initialize from a string or a natural number, the approach is to push_back from the end to the beginning. The primary distinction lies in the fact that, for string input, the numbers are converted to garbage characters since their ASCII code has been reduced by '0', corresponding to a subtraction of 48. The remaining two constructors are relatively simple, with one initializing from another UintN variable and the other initializing as an empty constructor, equivalent to 0.

It should be noted that the string constructor has an additional supporting function called "`string remove_leading(string str, char c)`" that was not included in this report. This function is designed to address the problem of leading zeros in the input number by removing them

from the string. It is important to mention that all of the string functions and other helper functions that were not covered in this report can be found in the code file.

The helper functions, as their name suggests, do not have a primary role in the required algorithms of the problem but are crucial in supporting the main functions. The name of each function clearly reflects its purpose and aids in maintaining the code's readability and organization.

```
// Cout
friend ostream& operator<<(ostream& out, const UintN& a)
{
    string s;
    for_each(a.BigInt.rbegin(), a.BigInt.rend(), [&](char c) {
        s.push_back(c + '0');
        });
    return out << s;
}
```

The operator<< is used for outputting the result, similar to cout in C++. In Python, the equivalent is the print() function. Both functions convert each digit back to its original ASCII code and either print it or add '0'. There is also a commented operator function in c++ code that work exactly like in Python version.

Note: In Python, there are certain helper functions and subsequent code sections that may not be implemented or optimized to the same extent as in C++.

Assignment, Comparision and Basic Math Functions:

These algorithms operate in a very simple way. For direct assignment, I simply delete the old number and assign the new one. The postfix and prefix operators, as well as post-decrement and pre-decrement, work exactly as they are implemented in the math libraries of C++. In Python, there is no equivalent to these operators, but they can be replaced with the +=, and -= operators.

```
//Assignment
UintN& operator =(const UintN& a)
{
    if (this != &a)
    {
        BigInt.erase();
        BigInt = a.BigInt;
    }
    return *this;
}
```

```
UintN& operator++();
UintN operator++(int temp)
{
    UintN aux = *this;
    ++(*this);
    return aux;
}
```

```
UintN& operator--();
UintN operator--(int temp)
{
    UintN aux = *this;
    --(*this);
    return aux;
}
```

The < operator works very simply. Firstly, it compares whether there is an equal number of digits in the two numbers or not. Then it loops through each digit, and if there is a difference between 2 corresponding digits, it compares which digit is larger. As you can see other operator will be based on the < operator, they work by just simply using logic of "!" and by swapping number a and b. The == operator will only need to compare the two strings with each other as well as != operator.

```
//Comparison
friend bool operator<(const UintN&, const UintN&);
friend bool operator<=(const UintN& a, const UintN& b)
{
    return !(a > b);
}
friend bool operator>(const UintN& a, const UintN& b)
{
    return b < a;
}
friend bool operator>=(const UintN& a, const UintN& b)
{
    return !(a < b);
}
Other comparision functions can be found in code file.
```

```
//Summation
UintN operator+(const UintN& b)
{
    UintN temp;
    temp = *this;
    temp += b;
    return temp;
}
UintN& operator+=(const UintN&);
```

```
//Subtraction
UintN operator-(const UintN& b)
{
    UintN temp;
    temp = *this;
    temp -= b;
    return temp;
}
UintN& operator-=(const UintN&);
```

```
Other basic math operator functions can be found in code file with details.
```

The basic math operator functions encompass addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). These operations adhere to the principles outlined on Geeks for Geeks. As demonstrated in the adjacent code snippet, all these operations share a common pattern. When performing addition or subtraction on large integers, the fundamental approach is applied. This involves adding the corresponding digits together and carrying over any resulting carry to the next digits, repeating this process until all digits have been accounted for. Likewise, multiplication follows the basic mathematical principle. Each digit of one number is multiplied by every digit of the other number, and the resulting products are summed together to obtain the final result. The division and modulus operations also adhere to basic mathematical principles.

However, the approach in the C++ code introduces some notable modifications to the operations. These include changes in the loop structures and the elimination of unnecessary variables, all aimed at optimizing the processing speed using LLVM. One significant adjustment is the utilization of size_t for loop handling, as size_t does not have negative values. Additionally, the inclusion of the magic number 4294967295 serves as a threshold for when a size_t variable reaches its minus value. Moreover,

instead of using vectors to store arrays in the division (/) and modulus (%) operators, pre-allocated arrays with fixed lengths and pre-assigned values are employed. These optimizations contribute to a significant boost in performance. And in Python version I have implemented power function in them, this function was missing in C++.

Miller-Rabin Primality Test:

Miller-Rabin is knowned as one of the most famous primality testing algorithms, also known as the strong pseudoprime primality test. The Miller-Rabin algorithm aims to determine whether a number is composite rather than proving it to be prime. It is an exceptionally fast algorithm that has successfully surpassed numerous Euler Pseudoprimes. However, it is important to note that certain strong pseudoprimes can still pass the test. The error rate of the Miller-Rabin test per iteration is $\frac{1}{4}$. Consequently, with k iterations, the error rate decreases to $\frac{1}{4^k}$.. Moreover, with the incorporation of several helper functions like deteting if that number is divisable for 2, 3, 5, 7, 11 lead to numerous of number will be detected and return false, so utilizing 5 iterations results in an error rate significantly smaller than 0.097%. The test works as follows:

- Suppose we have an odd integer n, such that $n = 1 + d \times 2^e$ and d is odd.
- We choose a positive integer $a < n$. If either $a^d \equiv 1\ (mod\ n)$, or $a^{2^r \times d} \equiv -1\ (mod\ n)$ for some $r < e$, then n is probably prime. Else, n is composite.

Example 1.2.1. Suppose $n = 65$. If we consider $a = 8$, we notice that $n = 1 + 1 \cdot 2^6$, $8^1 \equiv 8 \neq 1$, but $8^{2^{1 \cdot 1}} = 64 \equiv -1$. Thus, either 65 is prime, or 65 is a composite and 8 is a nonwitness. Of course 65 is not prime, but just to check, we consider $a = 2$. Clearly, $a^1 = 2 \not\equiv 1$, $a^{2^{0 \cdot 1}} = 2 \not\equiv -1$, $a^{2^{1 \cdot 1}} = 4 \not\equiv -1$, $a^{2^{2 \cdot 1}} = 16 \not\equiv -1$, $a^{2^{3 \cdot 1}} \equiv 61 \not\equiv -1$, $a^{2^{4 \cdot 1}} \equiv 16 \not\equiv -1$, $a^{2^{5 \cdot 1}} \equiv 61 \not\equiv -1$. Since 65 fails the Miller-Rabin Primality Test in base 2, we know that 65 is composite. We also know 2 is a witness to 65, but 8 is a nonwitness to 65.

Picture 1: Example of Miller-Rabin test from Shyam Narayanan Paper about Improving Speed and Accuracy of Miller-Rabins

Considering the inherent characteristics of the Miller-Rabin algorithm, the number of iterations required to test for primality can be quite high. To address this, I have devised an algorithm that leverages the multiprocessing method, enabling simultaneous testing of multiple numbers ($a^{2^r \times d}$) at once. This approach significantly enhances the efficiency of the primality testing process, as it allows for the concurrent evaluation of up to 12 numbers in a single iteration.

However, the algorithm optimizes the number of tests based on the number of digits in each number. Extensive research and experimentation have revealed that as the number of digits increases, a greater variety of different numbers a need to be tested. To achieve this, my algorithm utilizes a predefined list of numbers as test bases for positive interger a. This list comprises prime numbers below 50, carefully selected to ensure comprehensive coverage for primality testing.

```
void* ModularExponentiation(void* arg)
{
    struct arg4mod* a = (struct arg4mod*)arg;
    UintN n = a->n;
    UintN number = a->number;
    UintN exponent = a->exponent;

    if (n < 2)
        throw("Cannot perform a modulo operation against number
less than 2");
    if (number == 0)
        a->res = number;
    if (number >= n)
        number %= n;

    UintN res = UintN(1);
    while (exponent > 0)
    {
        if (exponent % 2 == 1)
            res = (res * number) % n;
        exponent /= 2;
        number = (number * number) % n;
    }
    a->res = res;
    pthread_exit(0);
}

struct arg4mod
{
    UintN number, exponent, n;
    UintN res;
};
```

Furthermore, I have employed a highly effective technique in C++ by utilizing void pointers to pass data parameters and retrieve the resulting values.

I have also implemented a fast modular exponentiation algorithm. This algorithm efficiently calculates the modular result of exponentiation, taking advantage of the properties of modular arithmetic. It allows for quick computation of large exponentiations modulo a given number, making it an essential component of the primality testing process.