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.

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.

```cpp
// 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.

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

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

```cpp
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                             //Subtraction
UintN operator+(const UintN& b)         UintN operator-(const UintN& b)
{                                       {
    UintN temp;                             UintN temp;
    temp = *this;                           temp = *this;
    temp += b;                              temp -= b;
    return temp;                            return temp;
}                                       }
UintN& operator+=(const UintN&);        UintN& operator-=(const UintN&);
Other basic math operator functions can be found in code file with
details.
```

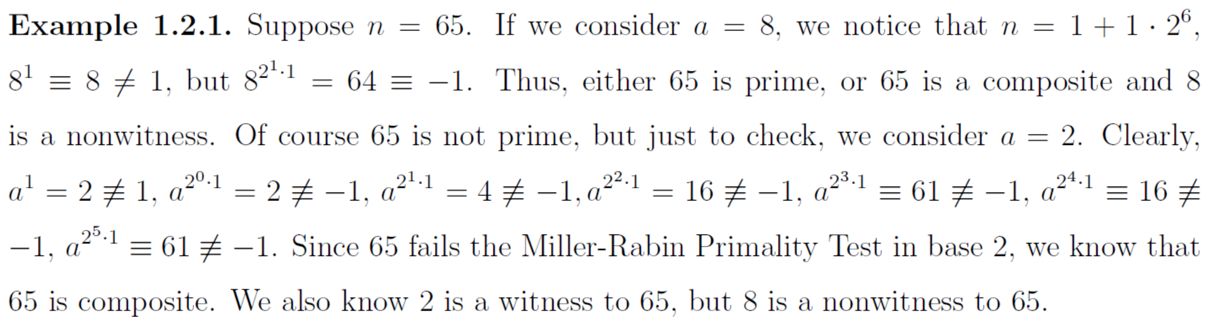
The basic math operator functions encompass addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). These operations adhere to the principles outlined on Geeks for Geeks. As demonstrated in the adjacent code snippet, all these operations share a common pattern. When performing 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 $a^d \equiv \pm 1\ (mod\ n)$, or $a^{2^r \times d} \equiv \pm 1\ (mod\ n)$ 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.

Code Breakdown:

1. ModularExponentiation:

```
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.

The ModularExponentiation function demonstrates the formidable prowess of the fast modular exponentiation algorithm. Notably, it harnesses the power of modular arithmetic to swiftly compute the modular result of exponentiation, enabling efficient handling of large exponentiations within the confines of a designated modulus. This algorithm's significance is particularly pronounced in primality testing scenarios, where numerous exponentiations modulo a specific number are required. Moreover, the ingenious utilization of the void* technique enhances its capabilities when working with pthread, enabling effortless argument passing and result retrieval despite the inherent complexities of pthread programming.

2. First Miller-Rabin Check Step:

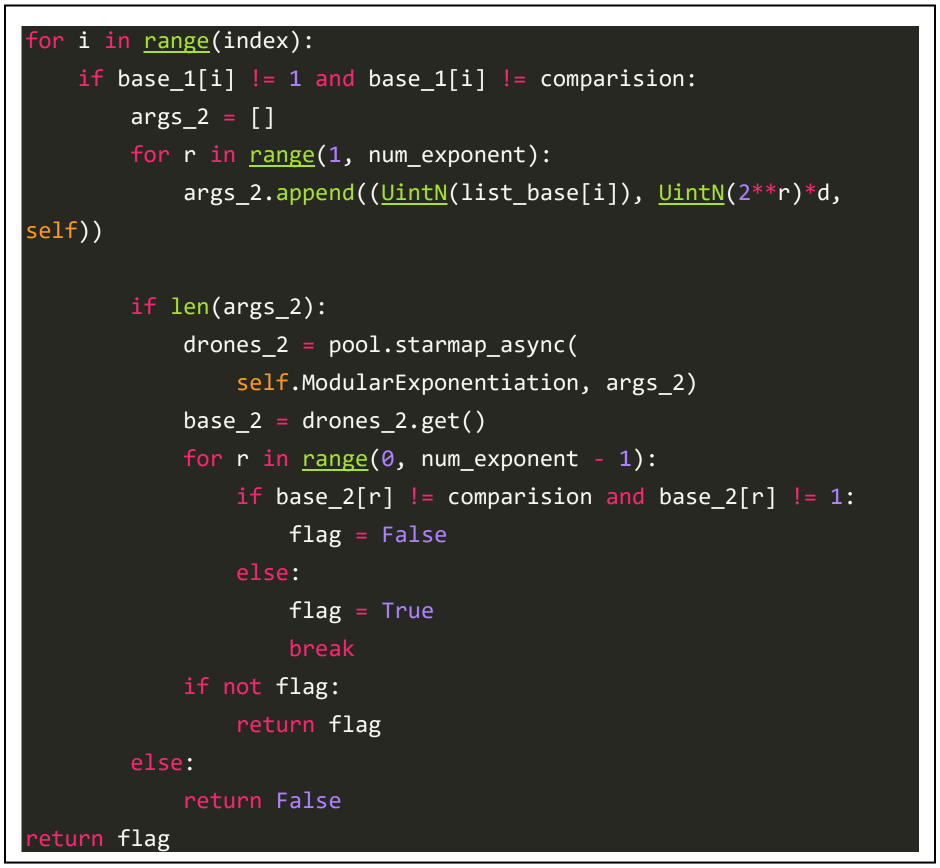
According to the Miller-Rabin algorithm's definition, it incorporates two key tests. The first test verifies whether $a^d \equiv \pm 1\ (mod\ n)$, and the second test involves a loop to examine the congruence $a^{2^r \times d} \equiv \pm 1\ (mod\ n)$ for each $r < e$. If either of these tests holds true, the number under scrutiny is highly likely to be a prime number. Hence, my implemented code follows a similar approach with two iterations, albeit with a slight variation. Leveraging the multiprocessing technique, I conduct the tests for multiple values of *d* based on the selected number of iterations (with a maximum of 5 iterations) of $a^d \equiv \pm 1\ (mod\ n)$.

```
struct arg4mod* args_1 = new arg4mod[index];
pthread_t* tids_1 = new pthread_t[index];
for (int i = 0; i < index; ++i)
{
    args_1[i].number = UintN(list_base[i]);
    args_1[i].exponent = d;
    args_1[i].n = number;

    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_create(&tids_1[i], &attr, ModularExponentiation,
&args_1[i]);
    pthread_attr_destroy(&attr);
}
for (int i = 0; i < index; ++i)
    pthread_join(tids_1[i], NULL);

//Find wrong indexes
vector<int> index_wrong;
for (int i = 0; i < index; ++i)
    if (args_1[i].res != 1 && args_1[i].res != comparision)
        index_wrong.push_back(i);
```

In the provided code snippet on the right, once the calculations for the numbers within the scope of a^d have been performed, the subsequent step involves verifying whether the obtained modulo results adhere to the condition $a^d \equiv \pm 1\ (mod\ n)$. If the computed results do not satisfy this criterion, the code proceeds to record the indices associated with the incorrect outcomes. This signifies the completion of the initial stage in the primality testing process. Subsequently, the algorithm progresses to the second phase, commonly referred to as "Step 2," where further examinations are conducted to ascertain the primality of the given number.

3. Second Miller-Rabin Check Step:

In the second step of the Miller-Rabin test, we calculate all the numbers of the form $a^{2^r \times d} \equiv \pm 1\ (mod\ n)$. As a result, there are considerably more numbers to compute compared to the first step. In this section, I will present two distinct approaches. While both approaches utilize multiprocessing, one method proves to be more resource-efficient in certain cases. Therefore, I will now proceed to discuss the first method in greater detail.



```
for i in range(index):
    if base_1[i] != 1 and base_1[i] != comparision:
        args_2 = []
        for r in range(1, num_exponent):
            args_2.append((UintN(list_base[i]), UintN(2**r)*d,
self))

        if len(args_2):
            drones_2 = pool.starmap_async(
                self.ModularExponentiation, args_2)
            base_2 = drones_2.get()
            for r in range(0, num_exponent - 1):
                if base_2[r] != comparision and base_2[r] != 1:
                    flag = False
                else:
                    flag = True
                    break
            if not flag:
                return flag
        else:
            return False
return flag
```

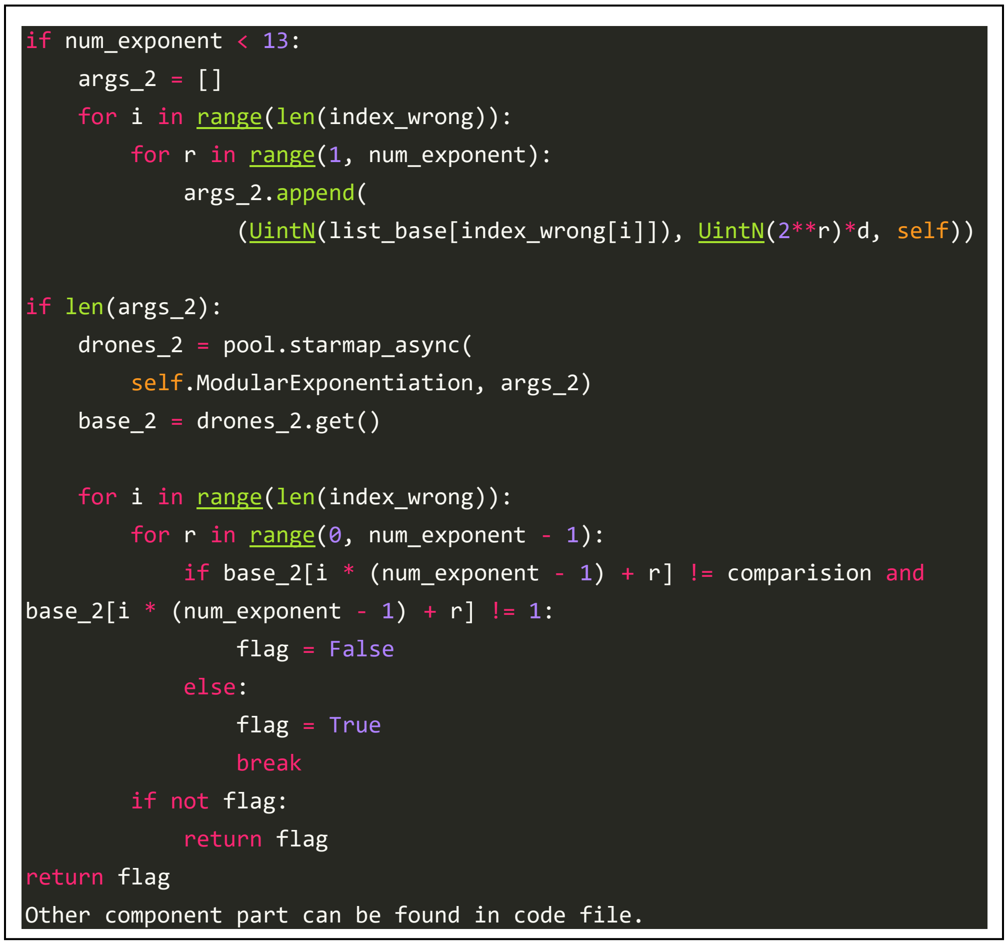
Here is the Python code snippet that implements the first method I mentioned. Its operation is straightforward. Its primary task is to handle each a^d value, and if any discrepancies are found, it saves all the corresponding numbers of the form $a^{2^r \times d}$ in memory for multiprocessing. It iterates through each number and checks if it meets the ±1 condition. If a satisfying condition is found, the loop is immediately broken, and the next number is processed. It continues checking until all r values have been examined. In such cases, it returns False. Additionally, if the initial a^d case does not meet the criteria and there are no further instances of $a^{2^r \times d}$, the function also returns False.

There are several disadvantages associated with this approache:

- When dealing with a large num_exponent (greater than the available CPU threads), if there is a ±1 result within the range, all the calculations must still be completed before performing the check. This leads to unnecessary residue computations.
- In the last iteration of the loop for each number a, there may be only a few r values (less than the available CPU threads) that need to be computed. This can result in the presence of idle threads, leading to extended processing time.

To avoid the aforementioned situations, the second method was introduced. This method can adjust dynamically based on the number of CPU threads available, ensuring optimal processing speed. This method comprises of two components. The first component focuses on processing numbers with a num_exponent ranging from 12 down to 1 (taking note that 12 represents the magic number representing the number of CPU threads). The second component is specifically designed to handle numbers with a num_exponent greater than 12. Due to the lengthy nature of the C++ code, it will not be included in this project. However, it can be found within the code file for reference. Instead, the Python code will be provided as a replacement since the approach for both methods is similar.

In the code snippet provided, you can see that all numbers of the form $a^{2^r \times d}$ with incorrect $a^{2^r \times d}$ results are grouped together for parallel computation, and the values of r are compared simultaneously. This algorithm guarantees that there will be, at most, one instance of idle CPU time.



```
if num_exponent < 13:
    args_2 = []
    for i in range(len(index_wrong)):
        for r in range(1, num_exponent):
            args_2.append(
                (UintN(list_base[index_wrong[i]]), UintN(2**r)*d, self))

if len(args_2):
    drones_2 = pool.starmap_async(
        self.ModularExponentiation, args_2)
    base_2 = drones_2.get()

    for i in range(len(index_wrong)):
        for r in range(0, num_exponent - 1):
            if base_2[i * (num_exponent - 1) + r] != comparision and 
base_2[i * (num_exponent - 1) + r] != 1:
                flag = False
            else:
                flag = True
                break
        if not flag:
            return flag
return flag
Other component part can be found in code file.
```

For numbers with a num_exponent greater than 12 (which surpasses the available CPU threads on my machine), a more straightforward approach is adopted. It involves running multiple loop iterations, and subsequently verifying the results. This method effectively addresses the concern of unnecessary residue computations, streamlining the overall process.

In order to further enhance the efficiency of the implemented methods, there are potential avenues for improvement. One such improvement could be the implementation of a worker pool in the C++ code. This approach involves creating a pool of reusable workers, rather than creating new workers for each task. By utilizing a worker pool, the overhead associated with worker initialization can be significantly reduced, resulting in improved resource utilization and overall performance. This optimization can contribute to a more efficient and streamlined execution of the code.

Results

Digits	**Speed**		
	C++	Python string	Python np.array
1->50	0.001-15s	0.1-60s	0.5-100s
180	41s	450-550s	~1000s
187	61s	450-550s	~1000s

Will be updated soon

References

Miller Rabin test:

- en.wikipedia.org/wiki/Miller-Rabin_primality_test
- www.geeksforgeeks.org/primality-test-set-3-miller-rabin/
- www.codeproject.com/Articles/60108/BigInteger-Library-2
- math.mit.edu/research/highschool/primes/materials/2014/Narayanan.pdf

Modular Exponentiation:

- homepages.math.uic.edu/~leon/cs-mcs401-s08/handouts/fastexp.pdf
- en.wikipedia.org/wiki/Modular_exponentiation

BigInt:

- www.geeksforgeeks.org/bigint-big-integers-in-c-with-example/
- github.com/WHU-Cryptography/BigInteger