# Lab 3

A prime number (or a prime) is a [natural number](https://en.wikipedia.org/wiki/Natural_number" \o "Natural number) greater than 1 that is not a [product](https://en.wikipedia.org/wiki/Product_(mathematics)" \o "Product (mathematics)) of two smaller natural numbers. A natural number greater than 1 that is not prime is called a [composite number](https://en.wikipedia.org/wiki/Composite_number" \o "Composite number). For example, 5 is prime because the only ways of writing it as a product, 1 × 5 or 5 × 1, involve 5 itself. However, 4 is composite because it is a product (2 × 2) in which both numbers are smaller than 4. Primes are central in [number theory](https://en.wikipedia.org/wiki/Number_theory" \o "Number theory) because of the [fundamental theorem of arithmetic](https://en.wikipedia.org/wiki/Fundamental_theorem_of_arithmetic" \o "Fundamental theorem of arithmetic): every natural number greater than 1 is either a prime itself or can be [factorized](https://en.wikipedia.org/wiki/Factorization" \o "Factorization) as a product of primes that is unique [up to](https://en.wikipedia.org/wiki/Up_to" \o "Up to) their order.

Prime numbers have a very important meaning in cryptography. For example, in the RSA algorithm, the security of the algorithm relies on the difficulty of decomposition of the multiplication of large prime numbers. In other words, the larger the prime number, the higher the security of RSA.

In this project, I used C++ to implement an algorithm for finding prime numbers.

*//*

*// findPrimes.h*

*//*

#ifndef FINDPRIMES\_H

#define FINDPRIMES\_H

class Solution {

public:

Solution();

static bool isprime(long *num*);

static char\* findPrime(long *start*, long *end*);

static double findPrimeMul(long *lim*);

};

#endif

#include "findPrimes.h"

#include <iostream>

#include <vector>

#include <string>

#include <sstream>

#include <chrono>

#include <thread>

using namespace std;

Solution::Solution() {}

bool Solution::isprime(long *num*)

{

long lim = *num*/2;

if(*num* == 1) {

return 0;

}

for(long i = 2; i <= lim; i++) {

if (*num* % i == 0) {

return 0;

}

else{ lim = *num*/i; }

}

return 1;

}

char\* Solution::findPrime(long *start*, long *end*)

{

vector<int> result;

for(long i = *start*; i <= *end*; i++) {

if(Solution::isprime(i)) {

result.push\_back(i);

}

}

stringstream ss;

string str;

copy(result.begin(), result.end(), ostream\_iterator<int>(ss, ","));

str = ss.str();

char \*p=(char\*)str.c\_str();

return p;

}

double Solution::findPrimeMul(long *lim*) {

vector<thread> threads;

auto start = chrono::high\_resolution\_clock::now();

for (int i = 1; i <= 4; i++) {

threads.push\_back(thread(findPrime, i, *lim*));

}

for (auto &th : threads) {

th.join();

}

auto end = chrono::high\_resolution\_clock::now();

double ret;

ret = chrono::duration<double, milli>(end - start).count();

cout << ret << " ms\n" << endl;

return ret;

}

#include "findPrimes.h"

#include <string>

#include <iostream>

#include <chrono>

#include <thread>

using namespace std;

void singleThread(long *number*) {

Solution sol;

auto start = std::chrono::high\_resolution\_clock::now();

auto end = std::chrono::high\_resolution\_clock::now();

std::cout << std::chrono::duration<double, std::milli>(end - start).count() << " ms\n";

cout << sol.findPrime(*number*) << endl;

}

int main() {

singleThread(100);

}

At the same time, I also implemented a multi-threaded version of the algorithm for finding prime numbers.

#include "findPrimes.h"

#include <iostream>

#include <vector>

#include <string>

#include <sstream>

#include <chrono>

#include <thread>

using namespace std;

Solution::Solution() {}

bool Solution::isprime(long *num*)

{

long lim = *num*/2;

if(*num* == 1)

{

return 0;

}

for(long i = 2; i <= lim; i++)

{

if (*num* % i == 0)

{

return 0;

}

else{ lim = *num*/i; }

}

return 1;

}

char\* Solution::findPrime(long *start*, long *end*)

{

vector<int> result;

for(long i = *start*; i <= *end*; i++)

{

if(Solution::isprime(i)) {

result.push\_back(i);

}

}

stringstream ss;

string str;

copy(result.begin(), result.end(), ostream\_iterator<int>(ss, ","));

str = ss.str();

char \*p=(char\*)str.c\_str();

return p;

}

double Solution::findPrimeMul(long *lim*) {

vector<thread> threads;

auto start = chrono::high\_resolution\_clock::now();

for (int i = 1; i <= 4; i++) {

threads.push\_back(thread(findPrime, i, lim));

}

for (auto &th : threads) {

th.join();

}

auto end = chrono::high\_resolution\_clock::now();

double ret;

ret = chrono::duration<double, milli>(end - start).count();

cout << ret << " ms\n" << endl;

return ret;

}

int main() {

auto start = chrono::high\_resolution\_clock::now();

Solution::findPrime(1, 1000000);

auto end = chrono::high\_resolution\_clock::now();

double ret;

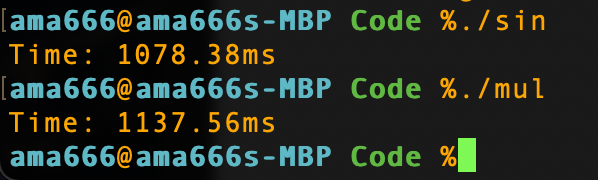
ret = chrono::duration<double, milli>(end - start).count();

cout << ret << " ms\n" << endl;

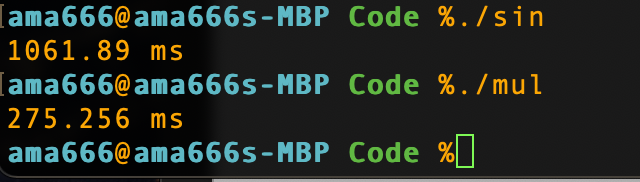
return 0;

}

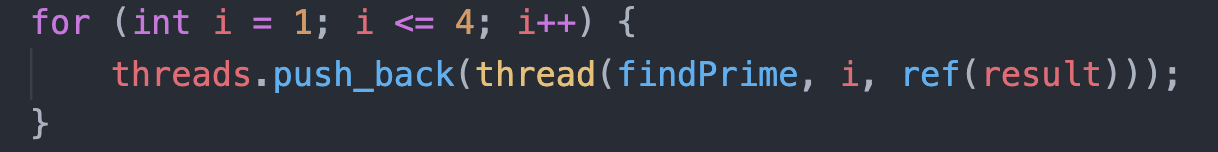
I used the clock function in the ctime library to calculate the running time of the two different versions, and found that the running time of the two are almost the same. Even the multi-threaded version runs slightly slower than the single-threaded version.

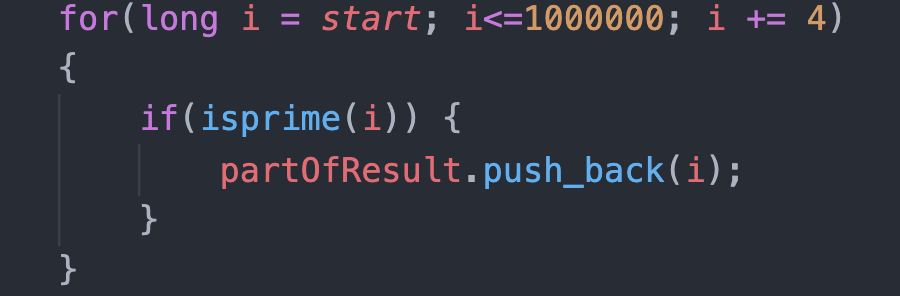


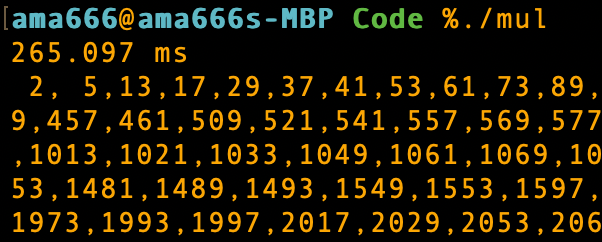
I searched the Internet and found that the clock function in the ctime library calculates CPU time. For multithreading, the CPU time of the program is fixed. I changed the clock function to the high\_resolution\_clock::now() function in the chrono library. As expected, the running time was greatly reduced.



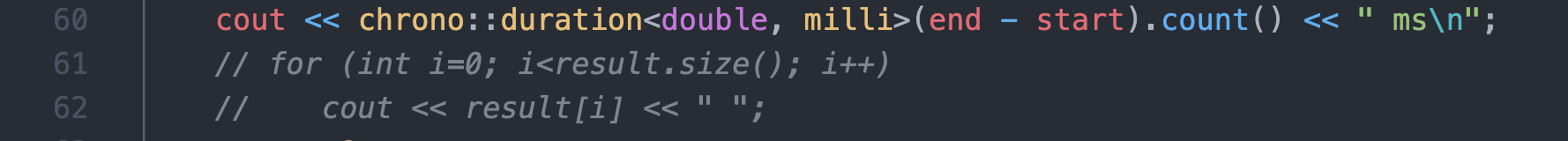
I use the multi-threaded version to find prime numbers within 1,000,000 and run it ten times. I set up 4 threads, and each thread will be responsible for a quarter of the workload.

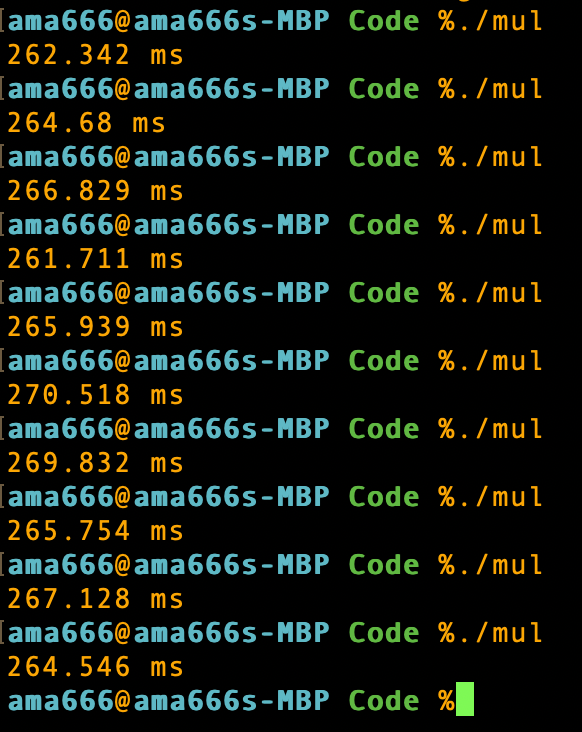




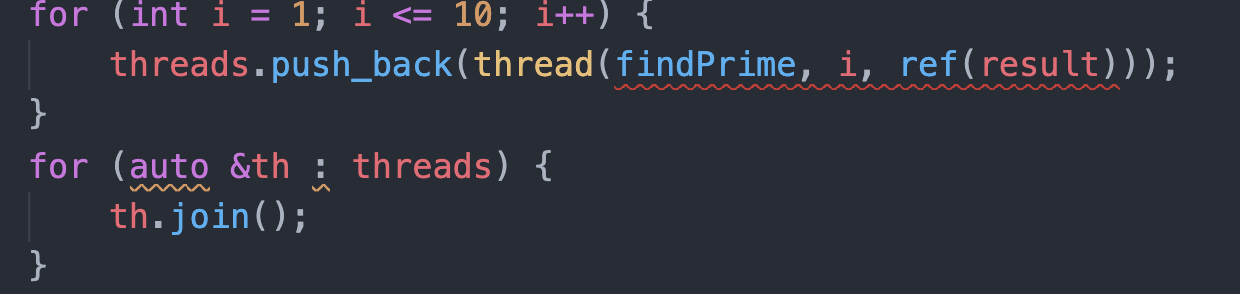


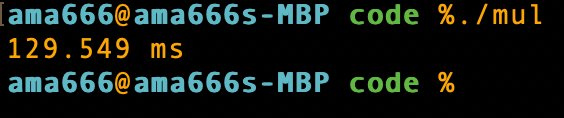
In order to facilitate the display of results, I only show the time-consuming of each round





When I increased the number of threads to 10, the speed doubled





Next, I will use a single thread to perform the same task and run it 10 times.



Now write a WebIDL interface file

interface Solution {

void Solution();

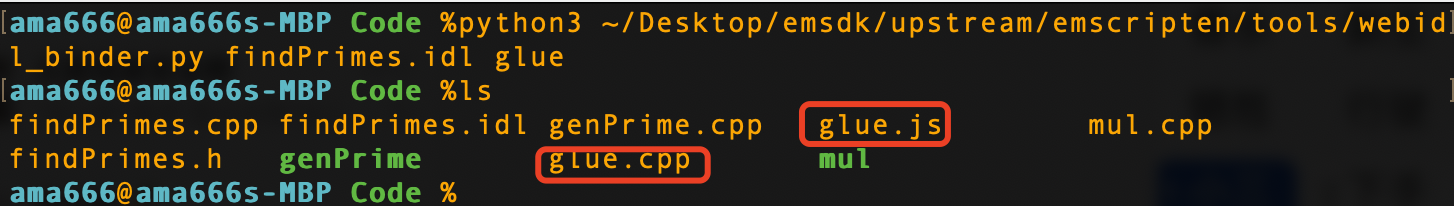
boolean isprime(long num);

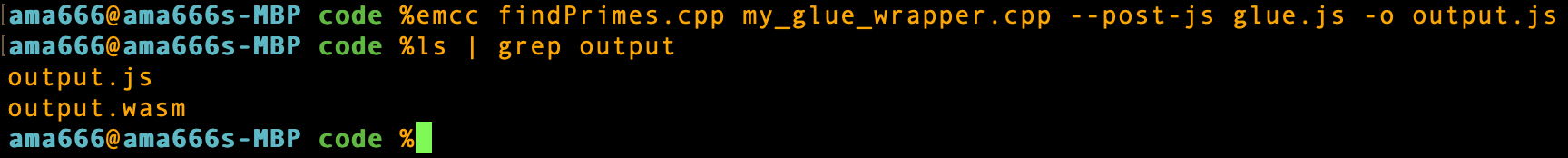
DOMString findPrime(long start, long end);

double findPrimeMul(long val);

};

**I have already explained some of the specific problems encountered in Lab2's report.**





Design a HTML page to show the result.

Durning this phase, I searched a lot how to do multithreading using Javascript. And all I got is javascript doesn’t support multithreading. So I just apply single thread version using javascript.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Get the Value of Text Input Field in JavaScript</title>

</head>

<body>

<p id="instruction">Type a integer, return prime number in 0 - integer.</p>

<input type="text" placeholder="Type something..." id="myInput">

<button type="button" onclick="getInputValue();">Get Value</button>

<p id="demo">The results will be shown here</p>

<script type="text/JavaScript" src="output.js"></script>

<script>

function insertAfter( *newElement*, *targetElement* ){

var parent = *targetElement*.parentNode;

if( parent.lastChild == *targetElement* ){

parent.appendChild( *newElement*, *targetElement* );

}else{

parent.insertBefore( *newElement*, *targetElement*.nextSibling );

}

}

function getInputValue(){

*// Selecting the input element and get its value*

var inputVal = document.getElementById("myInput").value;

var sol = new Module.Solution();

var lastObj = document.getElementById("demo");

for (var i=1; i<=10; i++)

{

let tStart = new Date().valueOf();

var result = sol.findPrime(1, inputVal);

let tEnd = new Date().valueOf();

let t = tEnd - tStart;

var p = document.createElement("p");

p.innerHTML = "Time: " + t + "ms\n" + result;

insertAfter(p, lastObj);

lastObj = p;

}

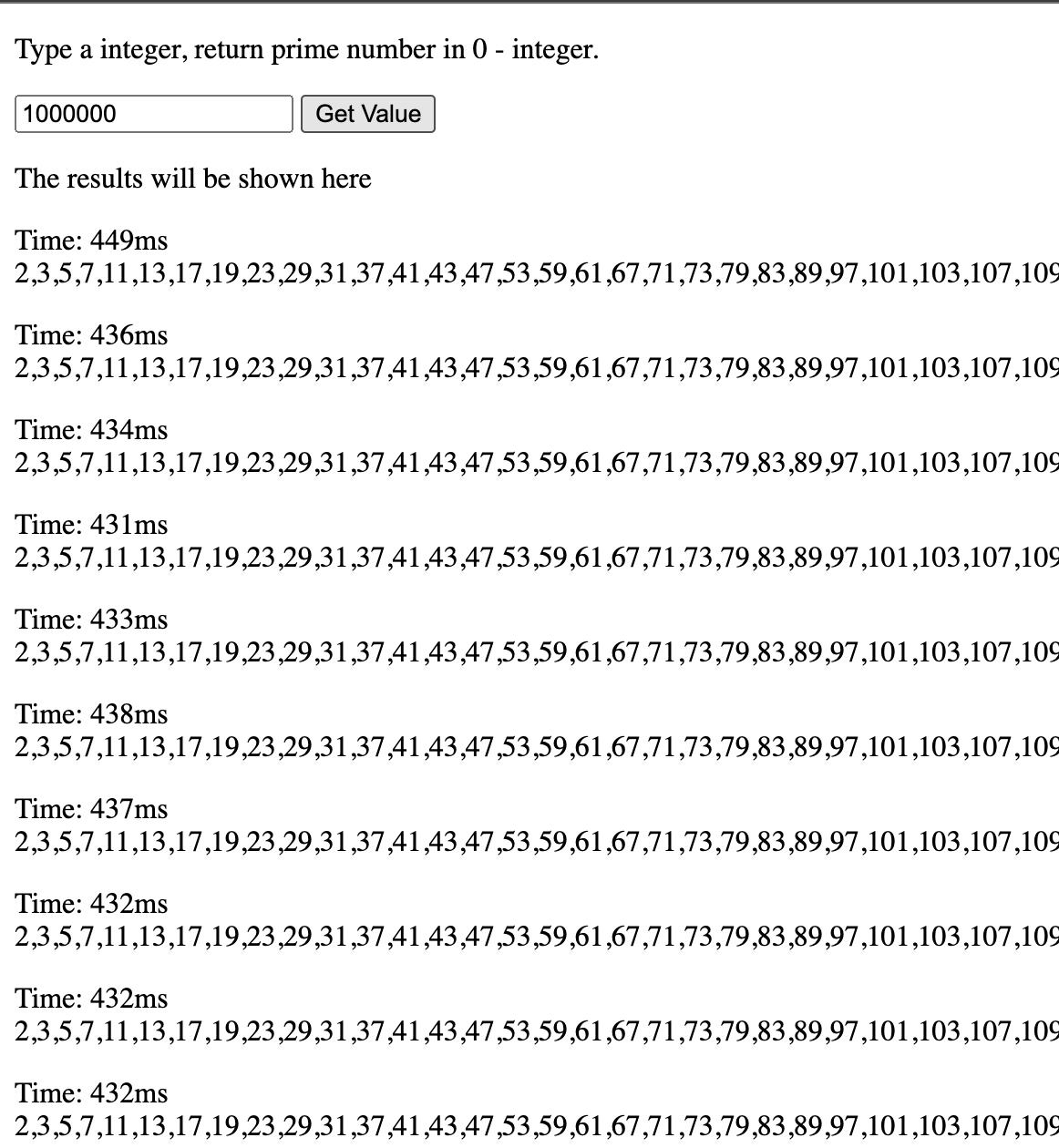
}

</script>

</body>

</html>

The result surprised me very much. According to past experience, the speed of javascript calling wasm to perform tasks must be slower than the speed of C++ native code to perform tasks. But for this task, using wasm to execute code is faster.



Time of Native C++ Single Thread:

Mean: 589.261

variance: 32.929

Standard deviation: 5.739

95% confidence interval: (585.704, 592.818)70% of data are in the confidence interval.

Time of Multithread C++ Single Thread:

Mean: 265.928

variance: 7.270

Standard deviation: 2.696

95% confidence interval: (264.257, 267.599)

60% of data are in the confidence interval.

Time of Single Thread in JS:

Mean: 435.4

variance: 25.640

Standard deviation: 5.064

95% confidence interval: (432.262, 438.538)

80% of data are in the confidence interval.