Physics 5 – Prime Numbers: Comparing Algorithms

In this project you will compare two algorithms for producing prime numbers.

The first program uses an inefficient algorithm, as shown in the code from lecture modified below:

```
//  Geoff Hagopian - CNSL003
//  A Function that Tests Primality

#include <iostream>

using namespace std;

bool isPrime(int);
// returns true if n is prime, false otherwise;

double sqrt(int); // returns the square root of an input integer
double abs(double x) return((x>0) ? x : -x);
// returns the absolute value of the input using a ternary operator

int main()
{ for (int n=0; n < 80; n++)
      if (isPrime(n)) cout << n << " ";
        cout << endl;
}

bool isPrime(int n)
{ // returns true if n is prime, false otherwise:
  double sqrtn = sqrt(n);
  if (n < 2) return false;           // 0 and 1 are not primes
  if (n < 4) return true;            // 2 and 3 are the first primes
  if (n%2 == 0) return false;        // 2 is the only even prime
  for (int d=3; d <= sqrtn; d += 2)
    if (n%d == 0) return false;      // n has a nontrivial divisor
  return true;                       // n has no nontrivial divisors
}

double sqrt(int x) {
      double tol = (double)x*1.e-17;
      double y = (double)x;
      double radicand = y; // the number whose square root we seek
      double sqrt = y/2;  // first guess
      while(abs(y - sqrt) > tol) {
            y = sqrt;
            sqrt = (y + radicand/y)/2;  // Babylonian algorithm
      }
      return(sqrt);
}
```

The best part of this code are the functions for computing the square root of an integer and the absolute value function. The isPrime function works, but is very inefficient .

Using the sieve of Eratosthenes algorithm is more efficient in many ways, especially as implemented in the C++ code implementing step 4 of the algorithm as laid out in FCC++ below:

The next application is an implementation of an algorithm that is attributed to the ancient Greek astronomer Eratosthenes of Cyrene (*c.* 276–194 B.C.). He was the first person to have calculated an accurate estimate of the circumference of the Earth.

Algorithm 5.1 The Sieve of Eratosthenes

To obtain a list of all the prime numbers (2, 3, 5, 7, ···) that are less than a given bound `max`:

1. Initialize an array `prime[max]` of `bool`s to be all `true` except the first two.
2. Set `prime[2*j] = false` for all `j > 1` for which `2*j < max`.
3. Set `prime[3*j] = false` for all `j > 1` for which `3*j < max`.
4. Repeat setting `prime[p*j] = false` for all `j > 1` for which `p*j < max` for each prime number `p` (= 5, 7, 11, *etc.*). The next prime will be the next `i` for which `prime[i]` is `true`.
5. When Step 4 is finished, the values of `i` for which `prime[i]` is `true` are the primes.

EXAMPLE 5.13 The Sieve of Eratosthenes

```
int main()
{ const int MAX = 1000;
  bool prime[MAX];
  prime[0] = prime[1] = false;
  for (int i=2; i<MAX; i++)
    prime[i] = true;
  for (int j=2; 2*j < MAX; j++)       // even numbers > 2 are not prime
    prime[2*j] = false;
  int p = 3;
  while (p <= MAX/2)
  { for (int j=2; p*j < MAX; j++)     // multiples of p are not prime
      prime[p*j] = false;
    do ++p
    while (!prime[p]);                // set p = next prime
  }
  for (i=2; i<MAX; i++)
  { if (prime[i]) cout << i << " ";   // print the primes
    if (i%80 == 0) cout << endl;      // avoid line wrap-around
  }
  cout << endl;
}
```

Note the effect of the `do` loop: it repeatedly increments `p` until `prime[p]` is true; *i.e.*, until `p` is a prime number again. Since `p` is already a prime when the loop begins, it must increment `p` before it evaluates `prime[p]`; so it must use the preincrement operator `++p`. The loop could also be written as

```
do
```

===

Your assignment is to write a program that allows the user to choose which of these algorithms to use and provide a way to compare the efficiency of one with that of the other. There are two good strategies for doing this that come to mind. Put in global counters to count the number of each operation as they are performed and then report on the total counts at the end, or use an actual timer that accesses the clock and compares the times to to find the nth prime in each.

Afterthought #0: Be sure your Win32 is not only empty, but that it's a "console application"

and not a Windows application. This caused us all a big headache in class last Thursday.

Afterthought #1: Since the console screen doesn't accommodate much output, if you want to see a listing of all the primes the program finds, write them to a file. To use the `outfile()` function include its library, `<fstream>`.

Afterthought #2: It's standard procedure to test the limits of a program . To determine the limits of the compiler you're using, include the library `<limits.h>`. Then set your local const int `MAXINT` in terms of the built-in `INT_MAX`. If `MAXINT` is too large your program may have troubles allocating memory. In the code below the program writes up to `INT_MAX/4096` to a file named "primes.txt". Note that you may have to go to projects directory and delete this file after running the program since it may fail to create a file that already exists.

```
int main()
{
	ofstream outfile("primes.txt"); //open file primes.txt for writing
	//int mult = 0, adds = 0, bitops;  // counters for op types
	cout << "\nINT_MAX = " << INT_MAX;  // << "UINT_MAX = " << UINT_MAX;
	const int MAXINT = INT_MAX/4096; //UINT_MAX/2;
	cout << "\nINT_MAX = " << INT_MAX;
	cout << "\nMAXINT = " << MAXINT;
	bool prime[MAXINT] = {true};  // This should initialize entire array
	for(int i = 0; i < MAXINT; i++) // …just to be certain…
		prime[i] = true;

	buildPrimes(prime, MAXINT); // use sieve of Eratosthenes
	cout << "\nThe primes less than " << MAXINT
	     << " are written to primes.txt";
	for (int i = 2; i<MAXINT; i++) {   // write the primes to a file
		if (prime[i]) outfile << i << " ";
		if (i%81 == 0) outfile << endl;
	}
	cin.get();
	return 0;
}
```

The output of this code is both to the file "primes.txt" and to the console screen:

```
INT_MAX = 2147483647
INT_MAX = 2147483647
MAXINT = 524287
The primes less than  524287 are written to primes.txt
```

Try modifying this code to use `unsigned int` types instead of `int` types and then `UINT_MAX` instead of `INT_MAX`. Then see how large you can make `MAXINT`.

Afterthought #3: To compare the efficiencies, we considered in class how to approximate FLOPS (floating point operations per second) or MIPS (millions of instructions per second) by counting operations in A simple option is to use the *uniform cost model* of time:

- One time unit per operation

There are Other considerations:

- On real machines, different operations typically take different times to execute. This will generally be ignored, but sometimes we may wish to count different types of operations, e.g., swaps and compares, or additions and multiplications.
- Some operations may be ``noise'' not truly affecting the running time; we typically only count ``major'' operations. For example, for loop index arithmetic and boundary checking is ``noise.'' Operations inside for loops are usually ``major.''

An algorithm solves an *instance* of a *problem*. There is, in general, one parameter, the input size, denoted by *n*, which is used to characterize the problem instance.

So we could use counters like `int bitops, adds, mults;` to measure increasingly more demanding operations and then insert increments of these into the code where they occur, such as in the labor-intensive square root function:

```
double sqrt(int x) {
      double tol = (double)x*1.e-17;
      double y = (double)x;
      double radicand = y; // radicand = number whose square root we seek
      double sqrt = y/2;  // first guess
      bitops += 6;   mults += 2;
      while(abs(y - sqrt) > tol) {
            y = sqrt;
            sqrt = (y + radicand/y)/2;
            mults +=2; adds += 2; bitops += 4;
      }
      return(sqrt);
}
```

Ultimately, time is best measured by a clock. To use the system clock, include the <ctime> library which allows you to create variables of type clock_t to set, say, start and end times.

The counter and time measures are moth implemented in the following main() function.

```
int main()
{
      //open filea primesFast.txt and primesSlow.txt for writing
      ofstream outfile1("primesFast.txt");
      ofstream outfile2("primesSlow.txt");
      // Determine the global limits and choose local limits proportionally
      cout << "\nINT_MAX = " << INT_MAX;
      const int MAXINT = INT_MAX/16256; //UINT_MAX/2;
      cout << "\nINT_MAX = " << INT_MAX;
      cout << "\nMAXINT = " << MAXINT;  //cout << "UINT_MAX = " << UINT_MAX;
      clock_t start, end; // create times for start and end
      long double duration;
      start=clock();
      bool prime[MAXINT] = {true};  //Should initialize the entire array
      for(int i = 0; i < MAXINT; i++) //but, to be sure...
            prime[i] = true;
      for (int n=0; n < MAXINT; n++) {
      if (isPrime(n)) outfile2 << n << " ";
```

```
            if(n%81 == 0) outfile2 << endl;
      }
      cout << "\n\nThe primes less than " << MAXINT
           << " are written to primesSlow.txt";
      end=clock();
      duration=(long double)(end-start)/CLOCKS_PER_SEC;
      cout << "\n\nInefficient algorithm time = " << duration << endl;
      cout << "\nbitops = " << bitops << "\nadds = " << adds
           << "\nmults = " << mults;
      bitops = 0; adds = 0; mults = 0;
      start=clock();
      buildPrimes(prime, MAXINT);
      cout << "\n\nThe primes less than " << MAXINT
           << " are written to primesFast.txt";
      for (int i = 2; i<MAXINT; i++) {   // write the primes to a file
            if (prime[i]) outfile1 << i << " ";
            if (i%81 == 0) outfile1 << endl;
            // do a carriage return every so often
      }
      end=clock();
      duration=(long double)(end-start)/CLOCKS_PER_SEC;
      cout << "\n\nEratosthenes algorithm time = " << duration << endl;
      cout << "\nbitops = " << bitops << "\nadds = " << adds
           << "\nmults = " << mults;
      cout << "\nThe primes less than " << MAXINT
           << " are written to primesFast.txt";
      cin.get(); // pause
      return 0;
}
```

Here is the output:

```
INT_MAX = 2147483647
INT_MAX = 2147483647
MAXINT = 132104

The primes less than 132104 are written to primesSlow.txt

Inefficient algorithm time = 0.141

bitops = 7362964
adds = 3285170
mults = 3549378

The primes less than 132104 are written to primesFast.txt

Eratosthenes algorithm time = 0.047

bitops = 408970
adds = 0
mults = 342885
```

Your task is to interpret the efficiency measures and to tweak this to push the limits as far as they'll go. Why is the bitops and mults off by a factor of 10 while the time is off by only a factor of 3?