Concepts of Parallel and Distributed Systems
CSCI-251

Project 2: Odd and Prime Number Generation

1 Goals

There are many tasks that are computationally expensive and can benefit from parallel execution. One such task is math problems that require brute force work. This could be anything from coming up with all possible values for a simple equation or attempting to break into a computer system that is protected with cryptography. We will be looking to reduce the run time of one such program using parallelism.

2 Overview

Write a program to generate large prime numbers and the factors of odd numbers using the C# parallel libraries.

3 Prime Numbers and Odd Numbers

Recall that a prime number is any number that is divisible by 1 and itself. Examples of prime numbers are 2, 3, 5, 7, etc. Mathematically speaking, the GCD(prime, any number) is always 1.

We will be generating prime numbers that are orders of magnitude larger, by generating a number and checking to see if it's prime, and only printing it to the console if it is. Odd numbers and their factor-count will also be generated,

In order to generate these large numbers, you will need to use the BigInteger class. Failure to use this class will cause your results to overflow and incorrect numbers will be generated.

The brute force approach to generating these large prime numbers will be to generate a large number and then check to see if it's prime. Because computers' random functions are generally very poor at being truly random, you should utilize the RandomNumberGenerator class in order to generate a random set of bytes. These bytes can then be used in the constructor of a BigInteger, which can then be checked for prime. The RandomNumberGenerator lives in the System.Security.Cryptography namespace. You MAY NOT use any other classes in this namespace other than the RandomNumberGenerator.

4 Requirements

- Your program must be a .net core command line program. The program should accept three arguments per the following help:

  dotnet run <bits> <option> <count>

  - bits - the number of bits of the number to be generated, this must be a multiple of 8, and at least 32 bits.
  - option - 'odd' or 'prime' (the type of numbers to be generated)
  - count - the count of numbers to generate, defaults to 1

- If the user specifies invalid command line arguments, you must print out an error message indicating the problem.

- When a prime number(s) is found, it should be printed to the console, along with the prime count(s), the number(s) and the time it took to generate all the numbers (using the Stopwatch class).
- When an odd number(s) is generated, it should be printed to the console along with the count(s), the number(s), the number of factors and the time it took to generate all the odd numbers(using the Stopwatch class). You do not need to list the factors.
- Your program should print the numbers as it finds them, not when it completes the specified count.
- Your program should NEVER crash.

5 Generating Factors of an odd Number

- You are free to use any algorithm with minimal time complexity to generate the factors of an odd number.
- Please take note that there may likely be an increase in the time it takes to generate these factors as the bit length increases!

6 Checking Prime

The following pseudo code may be used to check to see if a number is prime. You should implement an extension method with the following signature:

static Boolean IsProbablyPrime(this BigInteger value, int k = 10)

where k is the number of witnesses (10 is the default value and should work for this project). This method can then be called on a BigInteger as seen in the following pseudocode:

```
var bi = new BigInteger(32);
bi.isProbablyPrime();
```

You must write this method from the pseudo code below, you may not find a version on the Internet or a previous version of this project. You may use the ModPow function in your solution

```
Input #1: n > 3, an odd integer to be tested for primality
Input #2: k, the number of rounds of testing to perform
Output: "composite" if n is found to be composite, "probably prime" otherwise

write n as 2^r·d + 1 with d odd (by factoring out powers of 2 from n − 1)
WitnessLoop: repeat k times:
    pick a random integer a in the range [2, n − 2]
    x ← a^d mod n
    if x = 1 or x = n − 1 then
        continue WitnessLoop
    repeat r − 1 times:
        x ← x^2 mod n
        if x = n − 1 then
            continue WitnessLoop
    return "composite"
return "probably prime"
```

Figure 1: From: https://en.wikipedia.org/wiki/Miller-Rabin primality test

7 Design

- The program must be written using the C# parallel libraries
- The program must be command line driven
- The output (minus error messages), must match the writeup
- Command line help must be provided
- The program must take advantage (and run faster) on a multi-core processor
- The program must be designed using object oriented design principles as appropriate.
- The program must make use of reusable software components as appropriate
- Each public class/interface and property must include a comment describing the overall class or interface or property
- The parallelization should happen outside of the Miller-Rabin algorithm, that algorithm should not itself be parallelized. (see section 6)

8 Sample Runs (for output formatting)

I will test your project by running:

```
dotnet restore
dotnet run <bits><option> <count>
```

I will be testing your program on either a Mac or Windows machine, with secondary tests being run on Linux. You should ensure your program works on multiple platforms. Note that the numbers below are wrapped for display purposes only,

```
dotnet run 32 prime 2
BitLength: 32 bits
1: 1409777191

2: 180510569
Time to Generate: 00:00:00.0461866


dotnet run 128 prime 1
BitLength: 128 bits
1: 228879424573233456000131501845666751 69
Time to Generate: 00:00:00.0615736


dotnet run 32 odd 1
BitLength: 32 bits
1: 131073
Number of factors: 4
Time to Generate: 00:00:00.1955712
```

9 Grading Guide

This project will be graded over 100 points. I will grade your project by the following:

- (40 points) Evaluating the design of your program, as documented in the requirements and as implemented in the source code.
  - All of the Software Design Criteria are fully met: 40 points.
  - Code that is not thread safe in any manner will have a lose all 40pt deduction
- (10 points) Generating the factors of an odd number correctly. I will generate 4 tests. The first two tests will generate odd numbers with varying count, the other test will generate 2 single count odd numbers of varying bit length. Each test is worth 2.5 points
  - Correct number of Factors displayed: 2.5 points.
  - Incorrect number of Factors: 0 points.
- (20 points) Generating a prime number correctly. I will generate 4 tests; the first two tests will generate primes with varying count, the second test will generate 2 single count prime numbers of varying bit length. Each number is worth 5 points
  - Number displayed is prime: 5 points.
  - Number displayed is not prime: 0 points.
- I will grade these test cases based solely on whether your program produces the correct output as specified in the above Software Requirements. Any deviation from the requirements will result in a grade of 0 for the test case. This includes errors in the formatting (such as missing or extra spaces), incorrect uppercase/lowercase, output lines not terminated with a newline, extra newline(s) in the output, times printing incorrectly, and extraneous output not called for in the requirements. The requirements and sample runs state exactly how the output should appear on the console. Do not use a different output style. If any requirement is unclear, please ask for clarification.
  Note: you can lose points here and design criteria for the same flaw
- (30 points) Performance of generating 4 sets of prime numbers. Generating a valid prime can run in a variable amount of time, though anything less than 1024 bit keys should be able to complete in less than 1s on my machine (my solution has them typically completing in less than 0.1s for a single number). Failure to properly use a parallel algorithm and generating prime number too slowly can earn a deduction of up to 30 points. I will test up to 8192 bit numbers. Without adding logic to these larger numbers to improve performance, you will not get credit for the performance.

10 Submission Requirements

Zip up your solution in a file called project2.zip. The zip file should contain at least the following files with the specified names:

- Program.cs
- NumGen.csproj
- Text file if you use generative AI