PERSONAL PROJECT

EXPLORATION OF PRIME NUMBERS

Prime Number Display

Author
FRANK FENG

Supervisor
CHAU XIAN

Completion Date: January 21, 2024

Contents

List of Figures

List of Tables

0.1 Introduction

Prime numbers are like secret codes in the world of math. A total mysterious and undefined behavior of sequences and patterns. They're numbers that can't be divided evenly by anything other than 1 and themselves. In this paper, we will explore certain algorithms and clever arithmetic to obtain the n^{th} prime number. Additionally, the obtained prime number will be shown on a 4-digit seven-segment display powered by Arduino. This project combines mathematical proofs, electronics, and engineering creativity to showcase an interesting exploration of prime numbers. There will be three methods to obtain prime numbers including: C. P. Willans' prime factory, Sieve of Atkin, and Sieve of Eratosthenes.

0.2 Background Theory

A prime number is defined as a number whose only factors are 1 and itself. For instance, 11 is only divisible by 1 and 11. The smallest prime number is 2, making it the only even prime number. Notably, there is no formula for generating prime numbers, and it has been proven by Euclid that prime numbers continue infinitely. The infinite nature of prime numbers has been established, though this paper will not attempt to prove or define the specific pattern of prime numbers. Such a feat would a Millennium Prize Problem. This will make me a millionaire and also an award-winning mathematician by proving the Riemann hypothesis of prime numbers lying on the critical line.

Furthermore, it's crucial to distinguish between prime and composite numbers. While a prime number has no divisors other than 1 and itself, a composite number is characterized by having divisors beyond 1 and itself. This distinction is fundamental for understanding the various algorithms implemented in this project. However, there are further intricacies involving various caveats which will be discussed in the next section. Some of the methods take a unique approach, utilizing a process known as a sieve. This involves successively eliminating members of a list according to specific rules, leaving only certain elements. This process of filtering allows prime numbers to remain up to a given limit. Two notable methods are the modern Sieve of Atkin algorithm and the traditional Sieve of Eratosthenes algorithm.

0.3 Methodology

0.3.1 C. P. Willans' Formula

$$1+\sum_{i=1}^{2^n}\left\lfloor\left(\frac{n}{\sum_{j=1}^{i}\left\lfloor\cos^2\left(\pi\frac{(j-1)!+1}{j}\right)\right\rfloor}\right)^{1/n}\right\rfloor \tag{1}$$

This formula calculates the n^{th} prime number. For instance, the 1^{st} prime is 2, the 2^{nd} is 3, and so on. All components in this formula involve arithmetic and trigonometric procedures that have nothing to do with prime numbers on their own. However, the key to this formula lies in a separate theorem known as Wilson's theorem. Wilson's theorem states that the condition $(j-1)!+1$ is divisible by j precisely when j is prime or 1. In other words, $(j-1)!+1$ divided by j is an integer when j is prime and not an integer when j is composite. This property serves as a prime detector for Willans' formula. The table below illustrates this relation.

j	$\frac{(j-1)!+1}{j}$
1	2
2	**1**
3	**1**
4	$\frac{7}{4}$
5	**5**
6	$\frac{121}{6}$
7	**103**
8	$\frac{5041}{8}$
9	$\frac{40321}{9}$
10	$\frac{362881}{10}$
11	**329891**
...	...

Table 1: Wilson's Theorem table of values

We observe that when j is prime, Wilson's theorem produces an integer result. However, when j is not prime, the result is a fraction or simply not an integer. Willans' formula then proceeds with a trigonometric function, specifically the cosine of π. This allows the prime detector to output values in the range $[1,-1]$, and squaring it enables the detector to identify prime numbers by outputting 1 and composite numbers by outputting values in the range $[0,1)$. Furthermore, the floor function rounds down the given values, ensuring that the **prime detector (2)** signals 1 for prime numbers and only 0 for composite values. This mechanism is summarized by the equation below

$$\left\lfloor \left(\cos \pi \frac{(j-1)! + 1}{j} \right)^2 \right\rfloor = \begin{cases} 1 & \text{if j is prime} \\ 0 & \text{if j is composite} \end{cases} \tag{2}$$

Willans' formula proceeds with summing over a range of values using this prime detector, enabling a count of the amount of primes up to a given value. For instance, the summation from 1 to 10 will output $1+1+0+1+0+1+0+0+0+1$, which is equal to $4+1$. This indicates that there are 4 primes between the values 1 and 10. It's important to note that the $+1$ accounts for the integer value 1, which gives an integer result for the prime detector but is not a prime number. What we have now is a **numerical detector that counts the prime numbers (3)**. Although we know the count, we do not know the actual value of the desired prime. The summation is expressed as follows.

$$\sum_{j=1}^{i} \left\lfloor \left(\cos \pi \frac{(j-1)! + 1}{j} \right)^2 \right\rfloor = (\#\text{ primes} \leq i) + 1 \tag{3}$$

The next component of the formula involves having n in the numerator and taking the power of $1/n$. For simplicity, the denominator of the equation will be rewritten as follows:

$$\left(\frac{n}{(\#\text{ primes} \leq i) + 1} \right)^{1/n} \tag{4}$$

In order to understand this part, we can use an example. Let's assume we want to know the 4^{th} prime. Currently, we know that for numbers between 1-10, $(\#\text{ primes} \leq 10) = 4$, indicating that there are 4 primes. However, we want to know the actual value of the prime instead of just knowing how many there are. To achieve this, we will have to inspect the graph.

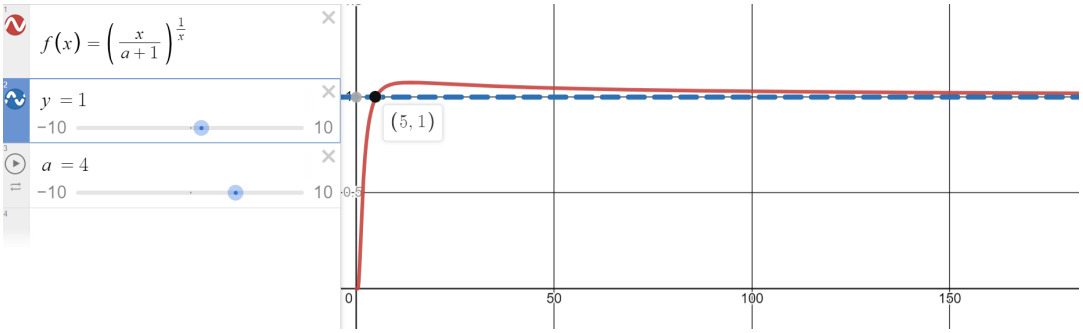

Figure 1: Graph of detector for numbers that are greater than 4.

From the graph, we observe that there is a clear distinction in the values before and after $x = 5$. The crucial detail to notice is that as x approaches positive infinity, y approaches 1. Consequently, by using the floor function, we now have a **detector for numbers greater than 4 (5)**. A minor detail to reiterate is that n is chosen for integer numbers, or non-decimal. The expression below explains the behavior from the graph.

$$\left\lfloor \left(\frac{n}{4+1} \right)^{1/n} \right\rfloor = \begin{cases} 1 & \text{if } n > 4 \\ 0 & \text{if } n \leq 4 \end{cases} \tag{5}$$

If we generalize this behavior for the n^{th} prime and its i value, then we have the following expression which is a **detector for numbers greater than i (6)**.

$$\left(\frac{n}{(\#\text{ primes} \leq i)+1}\right)^{1/n} = \begin{cases} 1 & \text{if } n > (\#\text{ primes} \leq i) \\ 0 & \text{if } n \leq (\#\text{ primes} \leq i) \end{cases} \tag{6}$$

Remember that for Willans' formula, the changing variable is i for a constant chosen n value. Therefore, we must refactor the conditions as follows leading to a **detector for numbers i that are less than the n^{th} prime (7)**.

$$\left\lfloor \left(\frac{n}{\sum_{i=1}^{n} \left\lfloor \left(\cos \pi \frac{(j-1)!+1}{j}\right)^2 \right\rfloor} \right)^{1/n} \right\rfloor = \begin{cases} 1 & \text{if } i < n^{\text{th}} \text{ prime} \\ 0 & \text{if } i \geq n^{\text{th}} \text{ prime} \end{cases} \tag{7}$$

The following component of Willans' formula involves another summation with a limit of 2^n. The limit is chosen based on Bertrand's postulate, which states that for every integer $m \geq 2$, there exists a prime number p such that $m < p < 2m$. The postulate guarantees a prime in the chosen interval, ensuring that there exist n primes between 1 and 2^n. For better understanding, let's consider the same example of finding the 4^{th} prime.

$$\sum_{i=1}^{2^4} \left\lfloor \left(\frac{4}{\sum_{j=1}^{i} \left\lfloor \left(\cos \pi \frac{(j-1)!+1}{j}\right)^2 \right\rfloor} \right)^{1/4} \right\rfloor = 1+1+1+1+1+1+0+0+0+0+0+0+0+0+0+0+0 = 6$$

There is a total of 16 numbers, which guarantees the 4^{th} prime. The detector outputs a 1 for each value of i that is less than the 4^{th} prime and a 0 otherwise. The summation takes charge of summing everything, leading to a value of 6, which is the 4^{th} prime minus 1. Therefore, the final step is to add 1 to the final **Willans' prime factory (8)** shown below.

$$1 + \sum_{i=1}^{2^n} \left\lfloor \left(\frac{n}{\sum_{j=1}^{i} \left\lfloor \cos^2 \left(\pi \frac{(j-1)!+1}{j}\right) \right\rfloor} \right)^{1/n} \right\rfloor = n^{\text{th}} \text{ prime} \tag{8}$$

Furthermore, the Arduino representation will contain a series of loops for the summation and a factorial function, as this is not a built-in functionality in the C program. The factorial function operates through recursion. The rest is just a matter of writing the formula properly. The code is shown as follows.

```
unsigned long factorial(int num) {
  // Factorial function using recursion
  if (num == 0 || num == 1) {
    return 1;
  } else {
    return num * factorial(num - 1);
  }
}

unsigned long nthPrimeWillans(int n) {
  unsigned long result = 1;

  for (int i = 1; i <= pow(2, n); i++) {
    float innerSum = 0.0;

    for (int j = 1; j <= i; j++) {
      innerSum += floor(pow(cos(PI * (factorial(j - 1) + 1) / j), 2));
    }
    result += floor(pow(n / innerSum, 1.0 / n));
  }
  return result;
}
```

Figure 2: Willans' formula in Arduino code.

0.3.2 Sieve of Eratosthenes

This sieve is developed in the 3rd century BC by Greek scientist Eratosthenes of Cyrene. He is known as the first person to calculate the circumference of the Earth by using the extensive results at the library. Additionally, his calculation was remarkably accurate. In number theory, which is what we are interested in, he develops the sieve of Eratosthenes which transverse the smallest prime number in the given range and eliminates multiples of the given prime number. This algorithm takes advantage of the fact that prime number cannot be expressed as a product of smaller integers, except for 1 and the number itself. The figure below illustrates a table that explains the procedure of the filtering process algorithm.

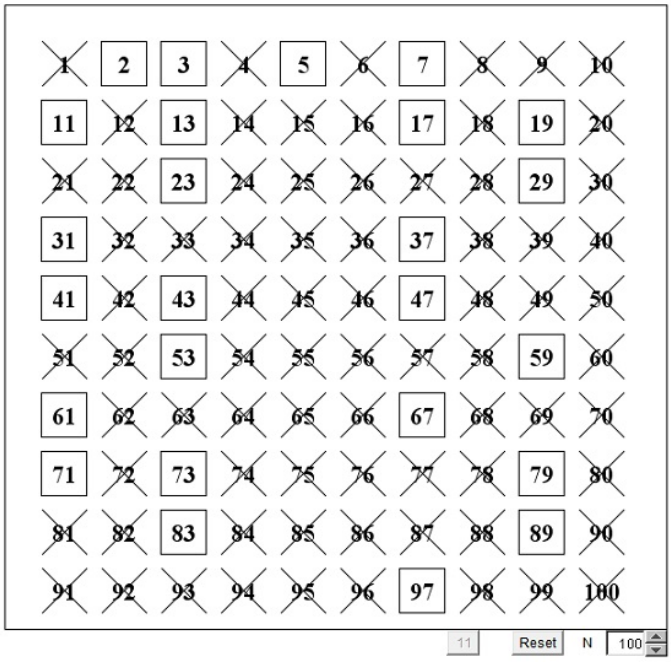

Figure 3: Sieve of Eratosthenes table.

The diagram illustrates how the Sieve of Eratosthenes identifies prime numbers. It begins with the first prime number, 2, and eliminates composite numbers divisible by 2. Consequently, all even numbers are crossed out. Moving on to the next prime number, 3, it eliminates composite numbers divisible by 3. As a result, all numbers whose individual digits add up to a multiple of 3 are eliminated. Notice that the composite number 6 is divisible by both 2 and 3, leading to its elimination twice. To avoid unnecessary eliminations, the preceding composite numbers are considered starting from the square of the prime number. For instance, the prime number 3 starts eliminating at 3 times 3, which is 9. The same pattern continues with 5, eliminating all numbers with a least significant digit of 0 or 5. Keep in mind that, in this case, the elimination of composite numbers starts from 25 onward. The process follows the same pattern, halting only when the selected prime number surpasses the specified range. For instance, with 11, the squared value of 121 exceeds the range of 1 - 100. Consequently, the remaining values that were not eliminated within the range are identified as prime numbers.

The Arduino code for this process is represented as follows:

```
void sieveOfEratosthenes(bool prime[], int n) {
  for (int p = 2; p * p <= n; p++) {
    if (prime[p]) {
      for (int i = p * p; i <= n; i += p)
        prime[i] = false;
    }
  }
}
```

Figure 4: Sieve of Eratosthenes function.

The Sieve of Eratosthenes algorithm utilizes a boolean array of size 'n' where 'true' represents a prime number and 'false' represents a composite number. The indices of the array correspond to the values of the numbers. The algorithm initializes the boolean array with all values set to 'true', assuming every number is prime:

$[T(0), T(1), T(2), T(3), T(4), T(5), T(6), T(7), T(8), T(9), T(10), T(11), T(12), T(13), T(14), T(15), T(16)...T(n)]$

It then begins an outer loop with 'p' starting from 2, the smallest prime number. The inner loop is executed only if 'p' is marked as prime. For example:

Given prime[2] is 'true', the inner loop starts by setting multiples of 2 to 'false': prime[4], prime[6], prime[8], prime[10], and so on. The resulting array becomes:

$[T(0), T(1), T(2), T(3), F(4), T(5), F(6), T(7), F(8), T(9), F(10), T(11), F(12), T(13), F(14), T(15), F(16)...T(n)]$

The algorithm then proceeds to the next prime, say 'p = 3'. The inner loop sets multiples of 3 to 'false', starting from prime[9] and continuing in increments of 3. The resulting array after this step becomes:

$[T(0), T(1), T(2), T(3), F(4), T(5), F(6), T(7), F(8), F(9), F(10), T(11), F(12), T(13), F(14), F(15), F(16)...T(n)]$

This process repeats until the square of the current prime is less than or equal to the specified limit 'n'.

```
unsigned int nthPrimeEratosthenes(int n) {
  const int MAX_SIZE = 8000;
  bool prime[MAX_SIZE];
  memset(prime, true, sizeof(prime));

  sieveOfEratosthenes(prime, MAX_SIZE);

  int count = 0;
  unsigned long result = 0;

  for (int p = 2; p < MAX_SIZE; p++) {
    if (prime[p]) {
      count++;
      if (count == n) {
        result = p;
        break;
      }
    }
  }

  return result;
}
```

Figure 5: Prime search with Sieve of Eratosthenes

This function begins by initializing a boolean array named 'prime' with a maximum size of 8000. While this size can be increased, it's crucial to consider the microcontroller's memory constraints. The 'memset' command is then utilized to set all values in the 'prime' array to true. Subsequently, the 'sieveOfEratosthenes' function is called and computed on the 'prime' array to identify all prime numbers.

The for loop starts with 2, which is the smallest prime number, and iterates through values from 2 to the maximum size. If 'prime' of the given 'p' is true after applying the Sieve of Eratosthenes function, the count increases. The count represents the number of prime numbers detected by the for loop or the number of "true" values looped through in the array. If the count equals the desired nth prime, the result is stored, and the loop exits with a 'break,' returning the result.

0.3.3 Sieve of Atkin

Developed in 2003 by mathematicians A. O. L. Atkin and Daniel J. Bernstein, the Sieve of Atkins is a modern algorithm compared to the Sieve of Eratosthenes. Its efficiency lies in some preliminary work done before the sieve process begins. Initially, it marks all possible numbers as non-prime or 'false,' except for 2 and 3. Utilizing modulus-sixty for each number, certain characteristics of the modulus are revealed to detect prime numbers. These characteristics give rise to the following quadratic form conditions.

1. Set true of given entry for each possible solution to $4x^2 + y^2 = n$ such that n % 12 is equal to 1 or 5.
2. Set true of given entry for each possible solution to $3x^2 + y^2 = n$ such that n % 12 is equal to 7.
3. Set true of given entry for each possible solution to $3x^2 - y^2 = n$ when x > y such that n % 12 is equal to 11.

For a clearer understanding, an example illustrating the rundown of numbers and operations will be provided in a table. Assuming a limit of $n = 25$, the conditions are checked as follows. All conditions must satisfy the requirement that n is less than the limit, in this case, 25.

$4x^2 + y^2 = n$	Flip Status	$3x^2 + y^2 = n$	Flip Status	$3x^2 - y^2 = n$ && $x > y$	Flip Status
$4(1)^2 + (1)^2 = 5$	Flip	$3(1)^2 + (1)^2 = 5$	No Flip	$3(2)^2 - (1)^2 = 11$	Flip
$4(1)^2 + (2)^2 = 8$	No Flip	$3(1)^2 + (2)^2 = 7$	Flip	$3(3)^2 - (2)^2 = 23$	Flip
$4(1)^2 + (3)^2 = 13$	Flip	$3(1)^2 + (3)^2 = 12$	No Flip		
$4(1)^2 + (4)^2 = 20$	No Flip	$3(1)^2 + (4)^2 = 19$	Flip		
$4(2)^2 + (1)^2 = 17$	Flip	$3(2)^2 + (1)^2 = 13$	No Flip		
$4(2)^2 + (2)^2 = 20$	No Flip	$3(2)^2 + (2)^2 = 16$	No Flip		
$4(2)^2 + (3)^2 = 25$	Flip	$3(2)^2 + (3)^2 = 21$	No Flip		

Table 2: Sieve of Atkin conditions example

The flipping status that determines whether a number is prime or composite is controlled by the conditions mentioned previously:

- Column 1: if (column1 value) % 12 = 1 or 5, then flip the sieve status for that number.
- Column 3: if (column3 value) % 12 = 7, then flip the sieve status for that number.
- Column 5: if (column5 value) % 12 = 11, then flip the sieve status for that number.

An additional condition must be checked. In the current example from Table 2, the status for 25 was flipped, marking it as prime. However, 25 is a multiple of 5 showing its composite feature. Therefore, for each prime number p less than or equal to the square root of the limit, all multiples of squares of the given prime number that are less than or equal to the limit are marked as non-prime. The sequence for this condition is $p^2 + ixp^2$ for $i = 0$ incremented by 1, continuing until the limit of n is reached.

Implementing the quadratic condition provided by Atkin follows the same concept as the boolean array with 'true' and 'false' conditions reflecting prime and composite numbers. This behavior was inspected in the Sieve of

Eratosthenes. The search for the n^{th} prime is the same for both sieve, incrementing a variable 'count' until it reaches the desired n. However, the approach to filter composite numbers differs. It uses the conditions shown in Table 2, signaling the prime status to be flipped. Each condition is implemented inside nested for loops, checking for each x and y value while simultaneously checking its modulo value for the given condition. The Arduino code for the Sieve of Atkin and the prime search is shown in Figures 6 and 7.

```
void sieveOfAtkins(bool *isPrime) {
  const int MAX_LIMIT = 8000;

  int sqrtLimit = static_cast<int>(sqrt(MAX_LIMIT));
  for (int x = 1; x <= sqrtLimit; x++) {
    for (int y = 1; y <= sqrtLimit; y++) {
      int n = (4 * x * x) + (y * y);
      if (n <= MAX_LIMIT && (n % 12 == 1 || n % 12 == 5))
        isPrime[n] = !isPrime[n];

      n = (3 * x * x) + (y * y);
      if (n <= MAX_LIMIT && n % 12 == 7)
        isPrime[n] = !isPrime[n];

      n = (3 * x * x) - (y * y);
      if (x > y && n <= MAX_LIMIT && n % 12 == 11)
        isPrime[n] = !isPrime[n];
    }
  }

  for (int x = 5; x <= sqrtLimit; x++) {
    if (isPrime[x]) {
      for (int y = x * x; y <= MAX_LIMIT; y += x * x) {
        isPrime[y] = false;
      }
    }
  }

  isPrime[2] = isPrime[3] = true;
}
```

Figure 6: Sieve of Atkin function.

```
int nthPrimeAtkins(int nth) {
  const int MAX_LIMIT = 8000;
  bool *isPrime = new bool[MAX_LIMIT + 1];
  memset(isPrime, false, (MAX_LIMIT + 1) * sizeof(bool));

  sieveOfAtkins(isPrime);

  int count = 0;
  for (int i = 2; i <= MAX_LIMIT; i++) {
    if (isPrime[i]) {
      count++;
      if (count == nth) {
        delete[] isPrime;
        return i;
      }
    }
  }
}
```

Figure 7: Prime search with Sieve of Atkin.

It is worth mentioning that the analysis of number theory and time complexity computation involved in these conditions is highly complex. I was unable to understand why these conditions work in such harmony to detect prime numbers so the detailed proof and analysis are presented in the original paper by Atkin and Bernstein, titled "Prime Sieves Using Binary Quadratic Forms."

0.4 Equipment

- MEGA 2560 Controller Board
- Arduino IDE
- 74HC595 IC
- 4-digit Seven-Segment Display
- Four NPN transistors PN2222
- Eight 220Ω and four 5$k\Omega$ resistors
- Two buttons
- A bunch of wires
- USB cable

0.5 Circuit Overview: Components, Code, and Schematics

0.5.1 Individual Components

The 74HC595 IC shift register is used to minimize port usage from the microcontroller, enabling sufficient control of the 4-digit seven-segment display with just three port connections. These connections include the shift register clock (SH_CP), data input (DS), and shift register latch (ST_CP), corresponding to pin positions 12, 14, and 11 on the IC, respectively. These three pins establish direct connections between the microcontroller and the shift register. The remaining pins serve purposes such as ground, power, and the resulting output of 8 bits sent to the display. The pinout configuration is illustrated in the figure below.

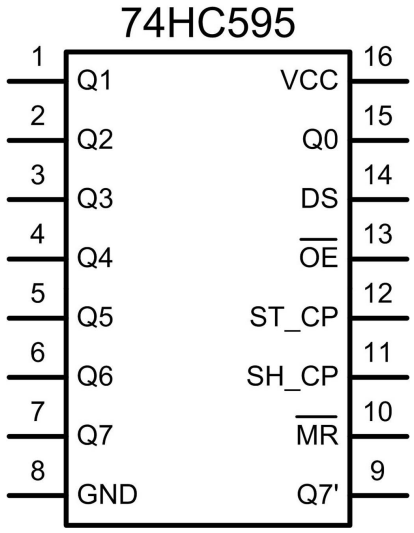

Figure 8: 74HC595 pinout configuration.

The shift register operates by sending individual bits to a storage register, controlled by the data input and the clock. The data input shifts one bit at a time, sending data to the storage register only when the clock trigger event occurs. The clock operates on a rising edge, meaning the trigger event happens when the signal transitions from logic 0 to logic 1. This behavior repeats until 8 bits, or an entire byte, are fulfilled. It's important to note that the first value shifted to the storage register is the least significant bit. Once the storage register is completed, the latch goes to logic 1, signaling the data to be sent to the display. An example of this mechanism is illustrated by sending the hexadecimal value $0x5F$.

Clock	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Data Input	1		1		1		1		1		0		1		0	
Latch	0		0		0		0		0		0		0		1	
Storage Register	0		1		0		1		1		1		1		1	
Result Pinout	Q7		Q6		Q5		Q4		Q3		Q2		Q1		Q0	

Table 3: 74HC595 shift registers example.

Sending each bit individually can be a tedious process, often involving multiple loops. To simplify this, the 'shiftOut' command enables the direct transmission of a byte of data without the need for extra loops. It's important to note that the latch still needs to be configured. The 'shiftOut' command takes four parameters as follows: 'shiftOut(dataPin, clockPin, bitOrder, value)'. The first two parameters are used for the data and clock, respectively. The 'bitOrder' specifies the start of the register, either starting from the Most Significant Bit (MSB) or Least Significant Bit (LSB). As a rule of thumb, if pin *Q7* is the MSB, then use MSB as the 'bitOrder' parameter. The last parameter represents where the data is taken from.

Furthermore, values from Table 3 are sent to the 4-digit seven-segment display, which has 12 pins. Eight of these pins are utilized to control the LED segments that form the displayed number, including the decimal points. The display is a common cathode, signifying that each digit's LED shares a common ground. Consequently, the same number is shown on all four digits given the same data input. For example, sending the number 5 will display "5555." However, there are four pins that enables turning on and off specific digits. Therefore, we can rapidly send the desired data to individual digits, turning them on and off fast enough to create the illusion that all LEDs are on at the same time. The pinout configuration for the 4-digit seven-segment display is shown below.

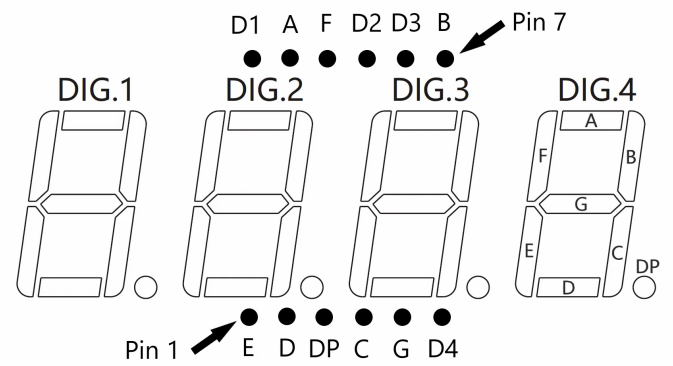

Figure 9: 4-digit seven-segment display pinout.

Having established the pinout configuration for both the shift register and the seven-segment display, we can determine the data arrangement in which the LEDs are set to obtain the desired number. For example, displaying the digit 8 is straightforward as all LEDs must be on, except the decimal point. However, displaying the digit 1 only requires LEDs B and C to be on. Therefore, it is necessary to create a table checking each LED value for each number from 0 to 9. The table, indicating each value and the corresponding pinout arrangement, is shown below.

Shift Register Pin	Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0	Hex
Segment Display Pin	G	C	DP	D	E	B	F	A	Value
0	0	1	0	1	1	1	1	1	0x5F
1	0	1	0	0	0	1	0	0	0x44
2	1	0	0	1	1	1	0	1	0x9D
3	1	1	0	1	0	1	0	1	0xD5
4	1	1	0	0	0	1	1	0	0xC6
5	1	1	0	1	0	0	1	1	0xD3
6	1	1	0	1	1	0	1	1	0xDB
7	0	1	0	0	0	1	0	1	0x45
8	1	1	0	1	1	1	1	1	0xDF
9	1	1	0	0	0	1	1	1	0xC7

Table 4: Display segment hexadecimal conversion

0.5.2 Arduino Code

From this point we will dive into the Arduino code using the data and theory covered before. First, the initialization for the pin configuration and the pin mode selection are shown on the figure below.

```
const int dataPin  = 12;  // 74HC595 pin 8 DS
const int latchPin = 11;  // 74HC595 pin 9 STCP
const int clockPin = 9;   // 74HC595 pin 10 SHCP
const int digit0   = 7;   // 7-Segment pin D4
const int digit1   = 6;   // 7-Segment pin D3
const int digit2   = 5;   // 7-Segment pin D2
const int digit3   = 4;   // 7-Segment pin D1
const int buttonIncreasePin = 2; // Increase button
const int buttonDecreasePin = 3; // Decrease button
```

```
void setup() {
    pinMode(latchPin,OUTPUT);
    pinMode(clockPin,OUTPUT);
    pinMode(dataPin,OUTPUT);
    pinMode(buttonIncreasePin, INPUT_PULLUP);
    pinMode(buttonDecreasePin, INPUT_PULLUP);
    for (int x=0; x<4; x++){
        pinMode(controlDigits[x],OUTPUT);
        digitalWrite(controlDigits[x],LOW);  // Turns off the digit
    }
}
```

Figure 10: Pins initialization and pin mode selection.

The global variables include the values from Table 4, arranged in an array where the indexes align with their corresponding display values. Additionally, the control digits array and the display digits array are crucial as they manage each individual digit. The control digits array handles the transistors that turn off/on the desired digit, while the display digits array contains the actual values of the digits. It's important to note that the digit at index 0 refers to the most significant digit, and the digit at index 3 refers to the least significant digit. The initialization of these variables is shown below.

```
unsigned char table[]=
    {   0x5F,  // = 0
        0x44,  // = 1
        0x9D,  // = 2
        0xD5,  // = 3
        0xC6,  // = 4
        0xD3,  // = 5
        0xDB,  // = 6
        0x45,  // = 7
        0xDF,  // = 8
        0xC7,  // = 9
    };
unsigned char digitDP = 32;  // 0x20 - adds this to digit to show decimal point
unsigned char controlDigits[] = {digit0, digit1, digit2, digit3}; // Control transistor
unsigned char displayDigits[] = {0, 0, 0, 0}; // Store actual display value number
unsigned int nth = 1; // Current nth prime
unsigned char debound = 0;
int brightness = 90;   // valid range of 0-100, 100=brightest
```

Figure 11: Global variables initialization.

The display function is a critical aspect, requiring the setting of each transistor in the appropriate order. The 'shiftOut' command and the latch allow the data to be sent to each digit using a for loop for each digit displaying and controlling it, ranging from 1 through 4. A delay of 1 microsecond is added between each digit. It's important to note that applying minimal delay will create the illusion that all digits are shown simultaneously, even though they are being displayed one digit at a time. The display function is illustrated in the figure below.

```
void DisplaySegments(int number){
     for (int i = 0; i < 4; i++) {
      // Get individual digit of number w/ modulo
       displayDigits[i] = table[static_cast<int>(number / pow(10, i)) % 10];
    }

    for (int x=0; x<4; x++){
        for (int j=0; j<4; j++){
            digitalWrite(controlDigits[j],LOW);    // turn off digits
        }
        digitalWrite(latchPin,LOW);            // Latch = 0, Enable Data
        shiftOut(dataPin,clockPin,MSBFIRST,displayDigits[x]); // Store Data
        digitalWrite(latchPin,HIGH);           // Latch = 1, Send Data
        digitalWrite(controlDigits[x],HIGH);   // turn on one digit
        delay(1);                              // 1 ms delay
    }
    for (int j=0; j<4; j++){
        digitalWrite(controlDigits[j],LOW);    // turn off digits
    }
}
```

Figure 12: Display number function.

The final step involves implementing the display with the sieve for prime numbers in a continuous loop. Three methods, including Willans' formula, Sieve of Eratosthenes, and Sieve of Atkin, all serve the same purpose. Using their functions to display the result completes the circuit. Additionally, two buttons are included to obtain the previous and the next prime numbers. After implementing a debounce, the loop will look as follows.

```
void loop() {

    // DisplaySegments(nthPrimeWillans(nth)); // Using Willans' Prime Factory
    // DisplaySegments(nthPrimeAtkins(nth)); // Using Sieve of Atkins
    DisplaySegments(nthPrimeEratosthenes(nth)); // Using Sieve of Eratosthenes

  if (digitalRead(buttonIncreasePin) == LOW && debound == 0)
  {
    nth++;
    debound = 1;
  }

  if (digitalRead(buttonDecreasePin) == LOW && debound == 0 && nth > 1)
  {
    nth--;
    debound = 1;
  }

  if (digitalRead(buttonIncreasePin) == HIGH && digitalRead(buttonDecreasePin) == HIGH)
  {
  debound = 0;
  }
}
```

Figure 13: Perpetual loop in Arduino code

0.5.3 Schematic

Here are the schematics of the circuit. I opted not to pay 8 euros for the Fritzing software, an open-source application for building circuits and schematics. However, credit for Figures 14 and 15 goes to Ricardo Moreno, who was working on a similar project involving a 4-digit hexadecimal counter. The only elements missing from his project are the two buttons for increasing and decreasing the value of n for the prime number. Other than that, the rest of the circuit remains the same and is depicted below.

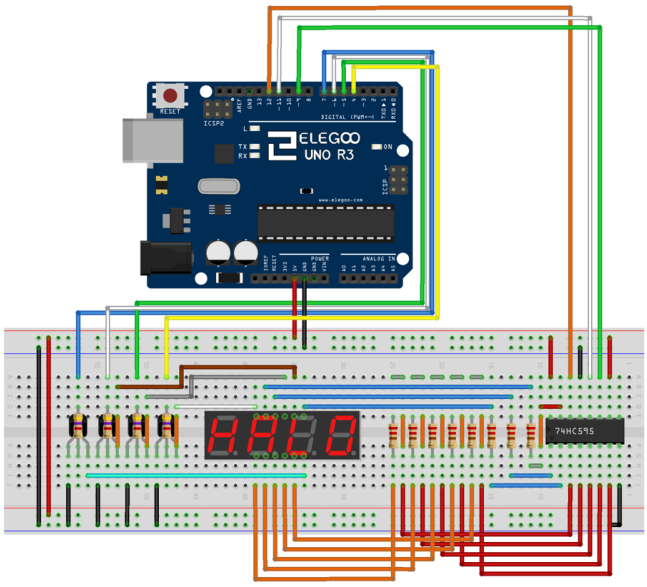

Figure 14: 4-digit seven-segment display circuit.

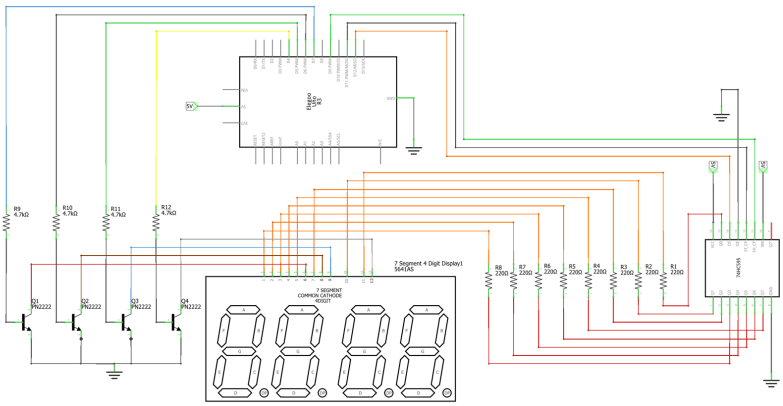

Figure 15: 4-digit seven-segment display schematic.

0.6 Results and Discussion

The circuit was assembled according to the provided schematic. Following the implementation of the prime number searching algorithm and the display function, the resulting circuit is presented below.

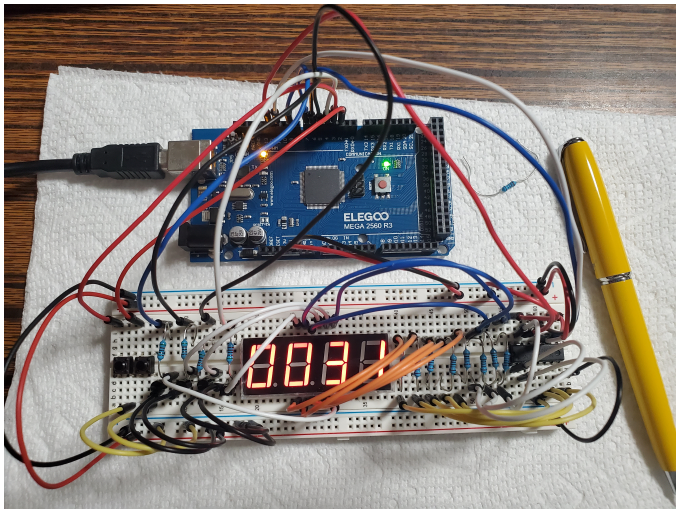

Figure 16: Image of finalized circuit.

Willans' formula is the least efficient method among the three. Its performance is not surprising, considering its multiple summations and factorial computations, resulting in heavy computational. Despite being a mathematical beauty in dealing with prime numbers, Willans' formula is impractical due to its poor efficiency. It successfully calculated the 3^{rd} prime number, which is 5, but proceeding attempts provided incorrect values such as 16 and 32. Further increasing n, the program failed to compute any number at all. On the other hand, Both the Sieve of Eratosthenes and Atkin methods performed very similarly. Despite Atkin being more modern and optimized, for a small project like this, there was no clear advantage of using one over the other.

The primary goal was to display prime numbers from 1 to 9999, considering the seven-segment having only 4 digits. However, the program encountered limitations and was unable to produce any prime numbers beyond 8000. Therefore, the maximum prime number displayed was the 1000^{th} prime, with a value of 7919. It is plausible that the issue lies in either memory constraints or computational challenges. Additionally, the optimization of the code itself could also be a factor, as it involves creating a large boolean array of 10000 items then searching for all prime numbers in a linear complexity.

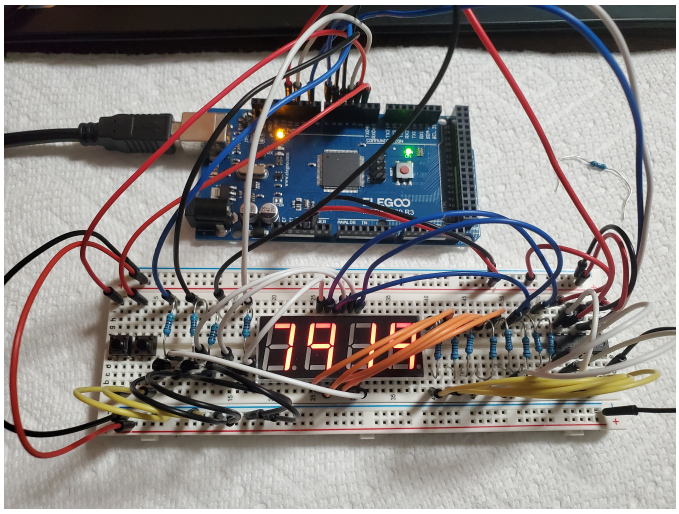

Figure 17: Circuit display largest possible value.

This project was both exciting and insightful, allowing me to explore into prime numbers and microcontroller applications. Future ideas could involve implementing interrupts instead of using debounce or creating a more optimal prime number search methods that can run in the background without interfering with the mainline code. The research journey was particularly interesting, exposing me to advanced topics such as the zeta function, the Riemann hypothesis, and coding patterns in algorithms, as demonstrated in the sieves used for this project.