NUMERICAL ADVENTURES WITH PRIME NUMBERS

Subash Bhusal
Eddie Federmeyer
Jamie Hayes
Vivek Ily
Nero Partida
Erik Rauum
Nicholas Ruffolo
Evangelos Kobotis

Introduction

The general purpose of this project was to explore the theory of prime numbers through a series of numerical experiments. Our main goals have been to study general and special properties of prime numbers and to understand the difficulties and limitations that one comes across while trying to translate the theory into actual usable information.

Our first task was to produce a list of prime numbers. To this end we came up with all prime numbers that do not exceed 10^8. This can be achieved by using quite classical means in a very small amount of time. This allowed us to start a further programming task by generating the primes in this range and then manipulating them as desired. In this way we were able to look for special kinds of primes, analyze their distribution and in general get acquainted with the speed that it takes to solve numerical problems like this in the range that we ended up referring to as the *trivial range*.

Our intention was to move outside the trivial range and explore things beyond its confines. The classical problem of finding large primes interested throughout the duration of this project. We did not have the ambition to find the next largest prime - that would have been pointless. We wanted however to find really large prime numbers, possibly prime numbers that have not been identified up to this point.

Simultaneously we sought to examine interesting and possibly more advanced parts of the theory of prime numbers. We looked at connections between the Prime Number Theorem and numerical applications. We engaged in a graphical exploration of the Ulam spiral. We looked at conjectural primality tests. We conducted numerical tests linked to the Dirichlet's theorem on primes in arithmetic progressions.

During our meetings everyone was engaged, bringing new ideas, discussing their successes and failures with the assignments of the previous week, volunteering for the new tasks or giving suggestions for different directions. The role of this presentation is to summarize some of the work that we did. In many cases we took code segments from the web but overall the code and results that we generated were our own work. We made every effort to give credit to our outside sources.

Prime numbers and the Eratosthenes sieve

We begin by looking at some standard notation and terminology. We will denote by $\mathbb{N}$ the set of natural numbers:

$$\mathbb{N} = \{1, 2, 3, 4, ...\}$$

and by $\mathbb{Z}$ the set of integer numbers:

$$\mathbb{Z} = \{0, 1, -1, 2, -2, 3, -3, \ldots\}$$

We say that the natural number n divides the integer number a if there exists $m \in \mathbb{Z}$ such that:

$$a = nm$$

We then write $n|a$ and we also say that a is divisible by n. It is evident that the number 1 is the only natural number that divides every integer number. On the other hand every natural number is divisible by itself and by the number 1. The natural numbers that are greater than 1 are then of two types. A natural number greater than 1 is called **prime** if it is divisible only by itself and the number 1. Otherwise it is called **composite**. It can be shown that every natural number greater than 1 is divisible by a prime number, or more precisely that it is either prime or a product of prime numbers. Such a representation is essentially unique and this is known as the Fundamental Theorem of Arithmetic.

We will also use the language of congruences. If n is a natural number then the we will say that the integer numbers are congruent modulo n if their difference is divisible by n. We then write:

$$a \equiv b \mod n$$

This is equivalent to a and b having the same remainder when they are divided by n.

An argument that goes back to Euclid shows that there exist infinitely many prime numbers. Indeed, give any nonempty finite set of prime numbers, then their product augmented by 1 is not divisible by any of those primes. Since it has to be divisible by at least one prime numbers, this establishes that there exists at least one prime number that does not belong to the initial finite set of primes. Hence no finite set of prime numbers can be exhaustive.

Exploring the properties of the set of prime numbers is one of the most fascinating and difficult tasks in mathematics. In this section we begin by looking over an ancient algorithm that theoretically produces all prime numbers. This is the Eratosthenes sieve and it is based upon the fact that every composite natural number is divisible by a prime number that does not exceed its square. Indeed if n is composite, then it can be written as ab where a and b are natural numbers greater than 1. If $a \geq b$, then $b^2 \leq n$ and if p is any prime number dividing b then $p^2 \leq n$ or $p \leq \sqrt{n}$.

This simple property has the following implication. If we start with the set:

$$\{1, 2, 3, \ldots, k^2\}$$

and we strike out 1 and all the numbers that are divisible by primes not exceeding k, then the numbers that are left behind are precisely the primes that are between k and k^2. In other words, if we know the primes that are less than k then we can easily find the primes that are less than k^2. More concretely, if we begin with the primes $2, 3, 5, 7$ that are precisely the primes that do not exceed 10, then we can easily find the primes that do not exceed 100. Once we can accomplish that, then we can easily find the primes not exceeding 10,000 and so on. Theoretically we can produce all prime numbers like this. However after a certain point the computations get so overwhelmingly long that this method proves to be rather inefficient for producing very large prime numbers.

We use this method in order to produce an initial segment of numbers that we call the *trivial range*. This consists of the primes that do not exceed 10^8. Note that before the computer age this would have been a quite remarkable achievement. However, with the emergence of computers this becomes a mere triviality. It gives us a satisfactory range of numbers to work with.

Let us now take a look at the programs that our team wrote:

We used an idea found in the website www.geeksforgeeks.org regarding a simple version of the Eratosthenes sieve. The following program was written by Subash Bushal. Its execution times determined what we considered as *the trivial range* for our work.

n-Value	# of Primes	Execution Time(s)
10000	1229	0.022
1000000	78498	0.235
10000000	664579	2.405
100000000	5761455	25.155

```

# sieve_prime.py - Subash Bhusal
# This program is to see our limiations of generating prime numbers on given constraints (time limit: 1
  hour)
#
# We are getting all primes up to a given n, and returning a list using sievePrime(n) method
# Then, we are using that list to make it into a pandas dataframe which is later converted into
# a csv file to get our list of prime numbers


import time # to time our program execution time
import pandas as pd # to create a dataframe and then convert to csv file

start_time = time.time() # to calculate our execution time

def sievePrime(n):

    # boolean list that intitalizes with all indices as true
    # value at prime[i] will be changed to false if i is a prime number

    prime = [True for i in range(n + 2)]
    p = 2
    while (p * p <= n + 1):
        if (prime[p] == True): # check if the value is already prime or not
    # updating all multiples of p in the given range
            for i in range(p * 2, n + 2, p):
                prime[i] = False
        p += 1

    # returns a new set that stores all prime numbers  from above algorithm
    prime_num = set()
    for p in range(2, n-1):
        if prime[p]:
            prime_num.add(p)
    return prime_num


# driver program
if __name__=='__main__':
```

```python
    # Calling the function
    n = 100000000 #100000000
    df = pd.DataFrame(sievePrime(n), columns=['Primes'])
    df.to_csv(path_or_buf='prime_numbers.csv')
    print(len(df))
    print("My program took", time.time() - start_time , "to run.")

```

Python code

The following program written by Vivek Ily implements the Eratosthenes sieve.

```python
import json


def prime_eratosthenes(n: int) -> list:
    # set for tracking composite numbers
    composite_set = set()

    # set for tracking prime numbers
    prime_set = set()

    # for all elements in the range n starting with first prime
    for i in range(2, n + 1):

        # if i is not a composite number
        if i not in composite_set:

            # add i to the prime set
            prime_set.add(i)

            # for all elements from i^2 to n+1 stepping i values at a time
            # by going i values each iteration, it adds only multiples of i to the composite set
            # any values that it skips will by default be primes
            for j in range(i * i, n + 1, i):
                composite_set.add(j)

    result = list(prime_set)
    result.sort()  # python doesn't like converting a set to a list and sorting on the same line

    with open("primes.json", "w") as write:
        json.dump(result, write)

    return result
```

Python code implementing the Eratosthenes Sieve

The following program was written by Nero Partida and it also implements the Eratosthenes sieve.

```python

def SieveOfEratosthenes(n):

  prime = [True for i in range(n+1)]
  p = 2
  while (p * p <=n):

      if (prime[p] == True):

      for i in range(p ** 2, n + 1, p):
        prime[i] = False
      p += 1
  prime[0]= False
  prime[1]= False

```

```python
  for p in range(n + 1):
         if prime [p]:
      print(p, end=' ')

if__name__= '__main__':
  n = 100000
  print("Following are thr prime numbers smaller ", end=' ' )
  print("than or equal to", n)
  SieveOfEratosthenes(n)
```

Python Code to generate Primes using the Sieve of Eratosthenes

Finally here is the code by Nicholas Ruffolo for the same task of identifying the primes in the trivial range. The sieve of Erastothenes is an algorithim that is capable of finding primes within a range of numbers by eliminating an integer if it is divisible by a smaller integer other than 1. In this case it is very effective to be ran on a significant, yet not entirely astronomical range of integers. In our research, we considered the range of 1-100,000,000. A primary consideration that was taken into account was the fact that the time it took to run the code at any 10^x greater than 10^8 resulted in hardware and time constraints unable to being to take care of the task. With the code above, and the aforementioned range, we had a run time of roughly under a minute (57.6 seconds).

```python

def sieve(n: int) -> list:
    """Sieve of Eratosthenes function. Give a value, n, and it will return a sorted list containing all
    primes in range [1, n+1]"""

    # set for tracking composite numbers
    composite_set = set()

    # set for tracking prime numbers
    prime_set = set()

    # for all elements in the range n starting with first prime
    for i in range(2, n+1):

        # if i is not a composite number
        if i not in composite_set:

            # add i to the prime set
            prime_set.add(i)

            # for all elements from i^2 to n+1 stepping i values at a time
            # by going i values each iteration, it adds only multiples of i to the composite set
            # any values that it skips will by default be primes
            for j in range(i*i, n+1, i):
                composite_set.add(j)

    result = list(prime_set)
    result.sort() # python doesn't like converting a set to a list and sorting on the same line
    return result
```

Python code

Special types and patterns of prime numbers

We look at special types of prime numbers. We begin with twin primes. Two primes are called twin primes if they differ by 2. In other words twin primes come in pairs. However, we can call a prime twin if it belongs to a pair of twin primes. There is only one prime number that belongs to two distinct pairs of twin primes: the prime 3. According to the twin prime conjecture, there exist infinitely many twin primes. To this date, this has not been proven or disproven.

We then look at Germain primes. An odd prime p is called a Germain prime if $2p+1$ is also a prime number. Germain primes originate in Sophie Germain's work on Fermat's last theorem. This was really the first attempt to attack Fermat's conjecture in a general way. Once more it is not known if there are infinitely many Germain primes or not.

A prime number is called **palindromic** if (in base 10) its representation is a palindrome. For example 101 is a palindromic prime.

A prime number p is called regular if its the class number of the number field $\mathbb{Q}(\zeta_p)$ is not divisible by p. This is a notion that requires one to go a little bit deeper from a theoretical point of view in order to understand it. However there are concrete ways to determine whether a given prime number is regular or not. Those ways go through the theory of the Bernoulli numbers. Bernoulli numbers can be defined by the formula:

$$\frac{x}{e^x - 1} = \sum_{n=0}^{\infty} \frac{B_n}{n!} x^n$$

It turns out that a number p is prime if and only if it does not divide the numerator of any of the Bernoulli numbers $B_2, B_4, \ldots B_{p-3}$.

A Mersenne prime is prime of the form $2^p - 1$ where p is a prime number. The largest prime numbers that have ever been found are Mersenne primes.

A permutable prime is a prime number that remains prime no matter how its digits are permuted. For example 13 is a permutable prime.

Another interesting topic is arithmetic progressions of prime numbers. The Green-Tao theorem suggests that there exist arbitrarily long arithmetic progressions among prime numbers. We would like to know how much we can say about the trivial range.

We also looked at gaps between prime numbers. It is a straightforward fact that the gap between two consecutive prime numbers can be arbitrarily large (none of the numbers between $n! + 2$ and $n! + n$ is prime for $n \geq 2$). We wanted to see what kind of gaps we expect to have in the trivial range.

Let's look now at the programs that were produced by our team.

The following program written by Jamie Hayes finds Germain primes in the trivial range. We also wanted to find *Germain sequences*, meaning finite sequences of primes so that each term is two times the previous term plus 1. It turns out that we have 14,156,112 Germain primes in the trivial range. The two longest Germain sequences within the trivial range start at 19099919 and 52554569.

```

def run(primes: list) -> dict:
  """
  Returns a dict of all Germain Prime sequences identified in the given list
  """

  # dict for storing results
  sequences = dict()
```

```python

    # building set of primes of O(1) checking
    primeSet = set(primes)

    # iterating through all primes in given list
    for prime in primes:

      # list for storing the current sequence achieved
      seq = list()

      # assigning the first prime to check as the current prime in the given list of primes
      gt = prime

      # checking that gt is Germain, and if so, adding to sequence and updating gt
      while (gt * 2) + 1 in primeSet:
        seq.append(gt)

        gt = (gt * 2) + 1

    # if the seq variable is not empty, meaning at least one Germain prime was identified, it gets added to results
    if seq:
      sequences[prime] = {"sequence": seq, "length": len(seq)}

    # returning the results
    return sequences


```

Algorithm to identify sequences of Germain primes.

The following program by Jamie Hayes looks for palindromic primes in the trivial range. It turns out that there exist 5,953 palindromic primes in the trivial range.

```python

  def eval_palindrome(prime: int) -> bool:
    """
    Checks if the given prime is a palindrome arithmetically.
    Returns boolean.
    """

    if prime < 10:
      return True

    # saving prime to variable n to check if n == reverse
    n = prime

    # variable to track reversed prime value
    rev = 0

    # since we are using floor division for prime, we iterate until prime <= 0
    while prime > 0:

      # the currect digit being worked on is the remainder from mod 10, giving us the last digit
      dig = prime % 10

      # multiply current reverse by 10 to allow for addition of dig
      rev = (rev * 10) + dig

      # floor division on prime to truncate last digit
      prime = prime // 10

    # if n, the starting prime, is equal to the reverse then it is a palindrome
    if n == rev:
      return True
```

```
    else:
      return False

```

Algorithm to identify palindromic primes

Here is a program written by Eddie Federmeyer that generates the Bernoulli numbers and stores the numerators as needed by Kummer's test:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from fractions import Fraction as Fr

"""
bernoulli_num(n) generates the first n bernoulli numbers and siores only the even bernoulli number
numerators (B(k) where k = 2n) in a file names bernoulli.txt. This script is a slightly modified
version of the code found at https://rosettacode.org/wiki/Bernoulli_numbers#Python. Personally, I
didn't have any use for the denominator of the bernoulli numbers so I simply ignore them. This is
much more optimal for space, however, the algorithm is still stunted in terms of speed since it's
practically exponential, taking nearly 9 hours to generate B(0) -> B(10,000) on an intel i7-6700k
"""

n = 1000    # Number of bernoulli numbers to generate.
            # The numerators wil be written to bernoulli.txt

def bernoulli_generator():
    A, m = [], 0
    while True:
        A.append(Fr(1, m+1))
        for j in range(m, 0, -1):
            A[j-1] = j*(A[j-1] - A[j])
        yield A[0] # (which is Bm)
        m += 1

def bernoulli_num(n: int, to_file = True) -> list[int]:
    bn = [ix for ix in zip(range(n), bernoulli_generator())]
    bn = [(i, b) for i,b in bn if b]

    if to_file:
        with open("bernoulli.txt", "w+") as file:
            index = 0
            for i, b in bn:
                # print('B(%2i) = %*i/%i' % (i, width, b.numerator, b.denominator))
                if (index > 1):
                    file.write("%s\n" % str(b.numerator))
                index += 1
    else:
        _bn = []
        for i, b in bn:
            _bn.append(b.numerator)
        return _bn

```

Code related to Bernoulli numbers and regular primes

Here is a program written by Eddie Federmeyer that implements Kummer's criterion for regularity:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import os

"""
```

```
This is a standalone script not meant to be called independently. This script reads
in all primes generated from sieve() and all bernoulli number numerators from
bernoulli_num(). It will spit out a new text file named regular.txt which contains
only regular prime numbers.

Regular Primes:
    Def => Prime numbers p which do not divide the numerator of any bernoulli number B(k)
    for all k where k = 2n and k <= p-3.

    For more info see: https://oeis.org/A007703
"""

# If neither file exists, abort!
if (not os.path.exists("bernoulli.txt") or not os.path.exists("primes.txt")):
    print("Please ensure that bernoulli.txt and primes.txt are in the root directory!")
    exit()

bernoulli_n = []
with open("bernoulli.txt") as file:
    for line in file:
        bernoulli_n.append(int(line))

primes = []
with open("primes.txt") as file2:
    for line2 in file2:
        primes.append(int(line2))

# Determine the largest prime which we are allowed to check!
p_3 = len(bernoulli_n)*2 + 3

regular_primes = []
for p in primes:
    if (p <= p_3):
        if (p-3 >= 0):  # This will exlude 2 since B(2-3) is not a valid bernoulli number
            is_reg = True
            # print(f"Checking if {p} is regular")

            # Check the divisibility of all even bernoulli numbers less than p-3
            index = 2
            for b in bernoulli_n:

                # Only check B(k) for 2, 4, 6, ..., p-3
                if (index > p-3):
                    break

                # If p divides B(k), than it is not a regular prime
                if (b % p == 0):
                    is_reg = False
                    break

                # This is just to keep track of p < p-3
                index += 2
        else:
            is_reg = False

        if (is_reg):
            # print(f"{p} is regular")
            regular_primes.append(p)

    else:
        break

# Dump it all to a file!
with open("reg.txt", "w+") as file:
    for reg in regular_primes:
```

```
            file.write("%s\n" % str(reg))

```

Code related to Bernoulli numbers and regular primes

The following program writen by Subash Bhusal counts twin primes. Here are the results depending on how many prime numbers we examine:

n-Value	# of Primes	Execution Time (s)
10000	205	0.004
1000000	8169	0.143
10000000	58980	1.66
100000000	440312	18.710

```

# twin_prime.py - Subash Bhusal

#
# This program uses the eratosthenes sieve algorithm to generate primes then use that list to check for
    all instances
# of twin prime numbers, prime numbers that differ by 2.
#
# Similiary to last program, we have recorded the execution time for this program as well as convereted
    the output into csv file to anayalze.

import time # to time our program execution time
import pandas as pd # to create a dataframe and then convert to csv file

start_time = time.time()
def twinPrime(n):

    # eratosthenes sieve - first we generate the primes using eratosthenes sieve,
    # where we set all values to true first, then set all non prime numbers as false
    prime = [True for i in range(n + 2)]
    p = 2
    while (p * p <= n + 1):
        if (prime[p] == True):
            for i in range(p * 2, n + 2, p):
                prime[i] = False
        p += 1

    #code for twin prime
    twin_prime = set()
    for p in range(2, n-1):
        if prime[p] and prime[p + 2]:
            twin_prime.add((p, p+2))
    result = list(twin_prime)
    result.sort()
    return result


# driver program
if __name__=='__main__':

    # Calling the function
    n = 100000000 #100,000,000
    df = pd.DataFrame(twinPrime(n), columns=['Prime A', 'Prime B'])
    df.to_csv(path_or_buf='twin_primes_fast.csv')
    print(len(df))
```

```python
    print("My program took", time.time() - start_time, "to run.")
```

Python code

Erik Raaum wrote the following program that searches the trivial range for arithmetic progressions of length 6.

```python

# arithmetic_progression.py - Erik Raaum
#
# arithmetic_progression holds one function, arithmetic_prog, which takes a list of primes and a maximum
    step size as inputs.
# The function returns a list of arithmetic progressions of length exactly 6.
# The first such series is {7, 37, 67, 97, 127, 157}.
# Checked up to 100,000,000 with step length up to 10,000 in 15.56 minutes: 348,120 series found

def length6(primes: list, max_step: int):

    prime_set = set(primes)
    count = 0

    # Iterate through all inputed primes.
    for prime in primes:

        # Iterate through each possible step size: Every sequence's step size will be a multiple of 30
        for step_size in range(30, max_step+1, 30):
            # sequence_list saves the sequence generated
            sequence_list = [prime]
            next_element = -1
            i = 1

            while next_element <= primes[-1]:

                # getting next in sequence
                next_element = prime + (step_size * i)

                # increment i
                i += 1

                # If next_element is a prime, we add it to the list
                if next_element in prime_set:
                    sequence_list.append(next_element)
                    # If the list has length 6, we stop and increment count.
                    if len(sequence_list) == 6:
                            count+=1
                            break
                else:
                    break

    return count
```

Python code

The following program written by Nero Partida looks for Mersenne Primes. What we find here is that in the first 100 million numbers we only get around 7 mersenne primes. Those are 3, 7, 31, 127, 8191, 131071, 524287.

```python

import math

def primes(n):

    sieve = [True] * (n // 2)
    for i in range(3, int(math.sqrt(n))+1, 2):
        if sieve[i//2]:
            sieve[i*i//2::i] = [False] * ((n - i*i -1) // (2*i) +1)
```

```
        return [2] + [2*i+1 for i in range (1, n // 2) if sieve[i]]

n = 1000

P = set(primes(n))

A = []
for i in range(2, int(math.log(n+1, 2))+1):
    A.append(2**i -1)

M = P.intersection(A)

print(sorted(list(M)))
```

Python code to calculate Mersenne Primes in a given range. Code by Nero Partida

The following code by Nicholas Ruffolo explored the prime gaps in the trivial range. It determined that the largest such gap is equal to 220 and it is between the numbers 47326693 and 47326913. The run time for the program was 45.1 seconds.

```

#Largest gap
def maxDelta(primes: list) -> dict:
    """Will take a sorted list of integers and determine the greatest gap between two consecutive values
    in the list. Returns a dictionary containing the the two values and the size of the gap."""

    # maximum is a variable for tracking the largest gap found. If a gap is greater than maximum, the
    value will be updated.
    maximum = 0

    # values is a variable for tracking the two numbers where the gap occurs.
    values = [0, 0]

    # this for-loop will iterate through all of the numbers in the list that is given as an argument
    # as we iterate through the list, we compare the number at the current index (i) to the number at the
     next index (i + 1)
    for i in range(0, len(primes) - 1):

        # n is a variable storing the number at index 'i'
        n = primes[i]

        # m is a variable storing the number at index 'i + 1'
        m = primes[i + 1]

        # delta is just the absolute value of the difference between n and m
        delta = abs(m - n)

        # if delta is greater than the current maximum gap, than:
        if delta > maximum:

            # maximum is updated to the new maximum gap
            maximum = delta

            # values is updated to store the two numbers where the gap occurs
            values = [n, m]

    # lastly, we return a dictionary which stores key:value valuess
    # 'primes' stores the two numbers were the gap occured
    # 'gap' stores the size of the gap
    return {'primes': values, 'gap': maximum }

```

Python code

The following program by Nicholas Ruffolo looked for permutable primes in part of the trivial range. It was based on code by Florian Rohrer as well as the sieve code written by Jamie Hayes. It did not find anything more than the following small permutable primes: 2, 3, 5, 7, 11, 13, 17, 31, 37, 71, 73, 79, 97, 113, 131, 199, 311, 337, 373, 733, 919, 991 and the execution time was about 11 minutes.

```

# permutable v2
from collections import deque
import itertools


def perms(p):
    result = []
    d = deque(str(p))
    for _ in range(len(str(p))):
        d.rotate(1)
        return [int(''.join(p)) for p in itertools.permutations(str(p))]
    return result

prev_primes = []

for poss_prime in range(2,1000):
    for n in prev_primes:
        if poss_prime % n == 0:
            break
    else: # no break
        prev_primes.append(poss_prime)

result = [p for p in prev_primes if all(q in prev_primes for q in perms(p))]
result.sort()
print(result)

#combined code to incorporate the Sieve
from permutablev2 import perms


def sieve(n: int) -> list:
    """Sieve of Eratosthenes function. Give a value, n, and it will return a sorted list containing all
    primes in range [1, n+1]"""

    # set for tracking composite numbers
    composite_set = set()

    # set for tracking prime numbers
    prime_set = set()

    # for all elements in the range n starting with first prime
    for i in range(2, n+1):

        # if i is not a composite number
        if i not in composite_set:

            # add i to the prime set
            prime_set.add(i)

            # for all elements from i^2 to n+1 stepping i values at a time
            # by going i values each iteration, it adds only multiples of i to the composite set
            # any values that it skips will by default be primes
            for j in range(i*i, n+1, i):
                composite_set.add(j)

    result = list(prime_set)
    result.sort() # python doesn't like converting a set to a list and sorting on the same line
    return result

prev_primes = []
```

```python

for poss_prime in range(2,1000000):
    for n in prev_primes:
        if poss_prime % n == 0:
            break
    else: # no break
        prev_primes.append(poss_prime)

result = [p for p in prev_primes if all(q in prev_primes for q in perms(p))]
result.sort()
print(result)
```

Python code

Dirichlet's theorem

Two integer numbers a and b are called coprime if and only if their only common divisor is the number 1. If a and b are natural coprime numbers, then according to Dirichlet's theorem, there exist infinitely many prime numbers of the form $an+b$. In fact it can be shown that for a given number a, the amounts of primes that correspond to the different remainders of division by a are uniformly distributed.

The following program was written by Erik Rauum, Subash Bhusal and Jamie Hayes and it computes the amounts of prime numbers with a given remainder.

n-Value	Average Proportion	Execution Time (s)
100000	0.025000000000000005	0.028
1000000	0.025	0.258
10000000	0.025000000000000005	2.689
100000000	0.025000000000000005	29.752

```

# dirichlet.py - Erik, Subash, Jamie
#
# This program examines the sequences q + nd, where d is coprime to q.
# It returns a dictionary where each entry q corresponds to its proportion
# of primes found.
#
# Got 100 up to 100,000,000 in 37 seconds.
# Accurate to 6 decimals.
# Got up to 1 billion in 48.88 minutes

from math import gcd
import numpy as np
import time

def runBool(n: int) -> list:
    """
    Sieve of Eratosthenes that uses a list of boolean values rather
    than integers in order to be more memory efficient.
    """
    bools = np.full(n + 1, True)
    bools[0] = False
    bools[1] = False
    i = 2  # Start at 2 since 0 and 1 are not primes

    while i * i <= n:
        # If prime[i] is not changed, then it is a prime
        if bools[i]:
            # Update all multiples of i as False
            for j in range(i ** 2, n + 1, i):
                bools[j] = False
        # Check next number
        i += 1

    results = list()
    for i in range(n + 1):
        if bools[i]:
            results.append(i)

    return results

# Converts a list of primes from sieve function to list of 1s for primes, 0s for composites
def sieve_to_binary(input):
    prime_binary = [0] * (input[-1] + 1)  # Create list of 0s
```

```
    for i in input:
        prime_binary[i] = 1  # Change prime numbered entries to 1
    return prime_binary


# This program examines the sequences q + nd, where d is coprime to q.
# It returns a dictionary where each entry q corresponds to its proportion
# of primes found.
# Got 100 up to 100,000,000 in 37 seconds.
# Accurate to 6 decimals.
# Got up to 1 billion in 48.88 minutes
def dirichlet_test(q):
    total_passed = 0  # Total primes of the form q+nd found
    prime_list = runBool(MAX_INT)  # List of primes (10,000,000)
    binary_list = sieve_to_binary(prime_list)  # Converts the list of primes to a binary list.
    primeLen = len(binary_list)
    proportions_dictionary = {}  # The dictionary to be returned

    # For any d coprime to q, we find the number of primes of the form q+nd and the total primes found
    for all d
    for d in range(q):
        if gcd(d, q) == 1:
            to_test = q + d
            d_test_passed = 0
            while to_test < primeLen:
                if binary_list[to_test] == 1:
                    d_test_passed += 1
                    total_passed += 1
                to_test += q
            proportions_dictionary[d] = d_test_passed

    # Convert a count of primes for each d value to a proportion
    for key, value in proportions_dictionary.items():
        proportions_dictionary[key] = value / total_passed

    # Returns the proportion of primes found in a dictionary
    return proportions_dictionary


if __name__ == "__main__":
    MAX_INT = 100000000
    value = int(input("Enter a value: "))
    start_time = time.time()
    D = dirichlet_test(value)
    # prints all the coprimes and their proportion
    for key, value in D.items():
        print(str(key) + ": " + str(value))
    print("My program took", time.time() - start_time, "to run.")
```

Python code

The Prime Number Theorem

The Prime Number Theorem states that the amount of prime numbers that does not exceed x is approximately $x/\log x$. One of the functions used in the process of proving the PNT is Chebyshev's theta function:

$$\vartheta(x) = \sum_{p \leq x} \log p$$

In fact the Prime Number Theorem itself is equivalent to the statement that:

$$\lim_{x \to \infty} \frac{\vartheta(x)}{x} = 1$$

It is also quite interesting that it has been proven that $\vartheta(x) - x$ changes sign infinitely many times. We wanted to explore this property by finding the sign of this expression in the trivial range.

Here is a program written by Jamie Hayes computing the function $\vartheta(x)$ in the trivial range. It turns out that $\vartheta(x)$ never changes sign in this range. In fact, one has to go to astronomically large numbers in order to observe the first change in sign.

```

from math import log


def run(primeList: list) -> list:
  """
  Chebyshev's Theta Function.
  Returns a sorted list containing the log transformed product of the primorial at each prime in the
 given list.
  """

  # list for storing the product at each nth prime
  products = list()

  # initiating var to store current Theta(x)
  lastVal = 0

  # iterating through all primes in the list
  for prime in primeList:

    # getting sum of logs
    # using log laws, we know log(n) + log(m) == log(nm)
    current = log(prime) + lastVal

    # storing to list
    products.append(current)

    # updating product
    lastVal = current

  return products


```

Computing Chebyshev's theta function.

Primality testing

Finding large prime numbers is one of the main goals in the theory of prime numbers. Here we apply a simple-minded search based on several theoretical results that we mention below. There are several primality tests. Some of them are probabilistic. Other are deterministic. Probabilistic primality tests identify numbers that have a strong probability of being prime. Deterministic tests, prove, when certain conditions are satisfied, that a given number is prime.

For some of this tests, it is useful to know Fermat's little theorem according to which if p is prime then for any integer a not divisible by p, we have:

$$a^{p-1} \equiv 1 \mod p$$

This means that if we have a number n for which the congruence:

$$a^{n-1} \equiv 1 \mod n$$

is wrong for a given a that is coprime to n, then it cannot be a prime number. On the other hand, if this is correct for a given a which is coprime to n, then this may make us hope (but certainly not decide) that n is prime. In fact this is the content of the **Fermat Primality Test**. One chooses a number a less than n and tests the equality $a^{n-1} \equiv 1 \mod n$. If it is true then we think of n as having some probability of being prime.

The **Miller-Rabin test** is also probabilistic and it is based on the following process. We have a number n and we consider a number a which is coprime to n. We then write $n - 1 = 2^s m$, where m is an odd number. We then test to see if the numbers

$$a^m, a^{2m}, \ldots, a^{2^s m}$$

are all equal to 1 with the possible exception of the first one that could be -1. If this is the case, then the test is passed by n and it has a strong probability to be a prime number.

It is conjectured that the following test is deterministic. If n is natural number satisfying:

- $2^{n-1} \equiv 1 \mod n$
- $F_{n+1} \equiv 0 \mod n$

then n is a prime number. Here F_{n+1} is the $n+1$-th term of the Fibonacci sequence defined by $F_0 = F_1 = 1$ and $F_{n+2} = F_{n+1} + F_n$.

The following program, written by Jamie Hayes, we are looking for big prime numbers using the following method. We choose a segment of natural numbers beyond the trivial range - say of length 10,000. We sieve out all multiples of prime numbers that do not exceed 10,000 and then we apply a probabilistic test like the Miller-Rabin test to the remaining numbers. Thus we get numbers that have a strong probability of being primes. To these we apply the Fibonacci primality test, which is conjectured to be deterministic.

```

from random import randrange
import numpy as np


```

```python
########## High Digit Sieve ##########
def sieve(sieve_primes: list, l: int, u: int, t: int) -> list:
  """
  sieve_primes: a list of trivial primes to sieve out majority of composite numbers in high digit range
  l: lower bound of interval
  u: upper bound of interval
  t: number of tests to iterate on each potential prime

  t=1 conducts single Fermat test
  t=60 for error rate of 2^(-128)
  """

  # ensuring boundaries are odd
  if l % 2 == 0:
    l -= 1
  if u % 2 == 0:
    u += 1

  shape = int((u - l) / 2)
  bools = np.full(shape, None)
  i = 0

  # iterating through only odd numbers on interval
  for n in range(l, u, 2):

    # initially assumed to be prime
    stat = True

    # checking if any prime factors exist for n
    # starts at smallest primes which are most probable
    for p in sieve_primes:

      # if p is a factor of n, we know its composite and exit check
      if n % p == 0:
        stat = False
        break

    # if all prime factor checks completed, run Miller-Rabin and store result
    if stat:
      bools[i] = _mr(n, t)
    # if prime factor checks failed, store result immediately
    else:
      bools[i] = stat

    # increment i
    i += 1

  # list of resulting primes
  primes = list()
  # iterator variable
  j = 0

  # cross checking elements in interval against boolean results
  for n in range(l, u, 2):

    if bools[j] == True:
    primes.append(n)

    j += 1

  # running each remaining prime through Fibonacci Primaility
  for p in primes:

    fib = _ft(p)

```

```
    if not fib:
      primes.remove(p)

  # returning result
  return primes


########## Miller-Rabin Test ##########
# This code has been duplicated from the millerRabin.py module to avoid requiring imports when sharing with the team.
def _mr(n: int, t: int) -> bool:
  """
  n: number to be evaluated
  t: number of test iterations
  Function for running the Miller-Rabin primality test.
  Will run the test t times.
  Returns a boolean showing that n is either composite or probably prime.
  """

  if n == 2 or n == 3:
    return True

  if n > 2 and n % 2 == 0:
    return False  # n is even

  # ensuring atleast 1 test iteration is run
  if t <= 0:
    t += 1

  # we will halve m iteratively until we achieve the equality:
  # n - 1 = (2^k)m
  k = 0
  m = n - 1

  # this loop halves m
  while m % 2 == 0:
    k += 1
    m //= 2  # floor div

  # we now have k, m such that m is an odd integer

  for _ in range(t):

    # determining random test value in range [2, n - 1]
    a = randrange(2, n - 1)

    # getting initial value for b
    b = pow(a, m, n)

    # initial check for primality
    if b == 1 or b == n - 1:
    continue

    # iterate until a result is found
    for _ in range(k - 1):

      # raising b**2 and getting remainder from modulo n
      b = pow(b, 2, n)

      # checking value of b
      if b == n - 1:
        break

    # if inner loop ends, check reason for ending
    # loop was broken
```

```python
      if b == n - 1:
        continue
      # loop ran out
      else:
        return False

    # if all iterations were completed without throwing False, then n is probably prime
    return True


  ########## Fibonacci Test ##########
  def _ft(n: int) -> bool:
    """
    Fibonacci test to identify if number is prime deterministically.
    Generates Fibonacci sequence modulo p.
    n: prime number
    """

    fib_seq = [0, 1, 1]
    i = 3
    while i <= n + 1:
      next_element = (fib_seq[-1] + fib_seq[-2]) % n
      fib_seq.append(next_element)
      del fib_seq[0]
      i += 1

    # Then we test if the n+1th term or the n-1th of the sequence is divisible by n
    # If yes, n is prime. Otherwise, n is composite
    if (fib_seq[-1] == 0) or (fib_seq[0] == 0):
      return True
    return False
```

Combination of several algorithms to sieve for primes in the non-trivial range, returning prime numbers deterministically.

Here is another program written by Vivek Ily that implements the Miller-Rabin test:

Another test used for primality testing is the Miller-Rabin primality test. Again, the Miller-Rabin primality test is a probabilistic primality test. The following code can be used for the Miller-Rabin primality test:

```python
from random import randrange

def miller_rabin(n: int, t: int) -> bool:
    """
    n: number to be evaluated
    t: number of test iterations
    """

    if n == 2 or n == 3:
        return True

    if n > 2 and n % 2 == 0:
        return False  # n is even

    # we will halve m iteratively until we achieve the equality:
    # n - 1 = (2^k)m
    k = 0
    m = n - 1

    # this loop halves m
    while m % 2 == 0:
        k += 1
        m //= 2  # floor div
```

```python

    # we now have k, m such that m is an odd integer

    for _ in range(t):

        # determining random test value in range [2, n - 1]
        a = randrange(2, n - 1)

        # getting initial value for b
        b = pow(a, m, n)

        # initial check for primality
        if b == 1 or b == n - 1:
            continue

        # iterate until a result is found
        for _ in range(k - 1):

            # raising b**2 and getting remainder from modulo n
            b = pow(b, 2, n)

            # checking value of b
            if b == n - 1:
                break

        # if inner loop ends, check reason for ending
        # loop was broken
        if b == n - 1:
            continue
        # loop ran out
        else:
            return False

    # if all iterations were completed without throwing False, then n is probably prime
    return True
```

Python code to determine primality (probabilistically) by using the Miller-Rabin primality test

Here is a program written by Vivek Ily that implements Fermat's test:

```python
from random import randint


def little_theorem(n: int, k: int) -> bool:
    # These are easily determined, no need for the flt
    if n < 2:
        return False
    elif n == 2 or n == 3:
        return True
    elif n % 2 == 0:
        return False

    for _ in range(k):
        a = randint(2, n - 2)

        if pow(a, n - 1, n) != 1:
            return False

    # if all iterations were completed without throwing False, then n is probably prime
    return True
```

Python code to determine primality using the Fermat primality test

The following program was written by Erik Rauum and it implements the Fermat primality test:

```python

# exponent_generator.py - Erik Raaum
#
# This is a function to generate a list of numbers between inputs m and n,
# so that for each number i in the range, i divides  2^(i-1)-1.
# While each returned number is prime, it should be noted that this test does not find all primes in the input range.
# In testing, we checked between 1,000,000,000 and 1,000,000,050 in 27 minutes.
# The numbers found were 1,000,000,007, 1,000,000,009, 1,000,000,021, and 1,000,000,033.


def exponent_generator(m,n):
    # Passed is a list storing the numbers that pass the test.
    passed=[]
    # to_test is the number that will be passed through the test next.
    to_test=m
    while to_test<=n:
        # Quickly sieve out some numbers
        if to_test % 2 == 0 or to_test % 3 == 0 or to_test % 5 == 0:
            to_test+=1

        else:
            # Here we check if to_test divides 2^(to_test-1).
            # To calculate 2^(to_test-1) quickly, we do so modulo to_test.
            c = 1
            for i in range(1,to_test):
                c = (2*c) % to_test
            # If so, we add the number to passed.
            if c == 1:
                passed.append(to_test)
            to_test+=1

    return passed
```

Python code

The following program was written by Erik Rauum and it implements the primality test that is based on the Fibonacci sequence.

```python

# Fibonacci_test.py - Erik Raaum
# This function runs the Fibonacci primality test on a single number to determine its primality.
# Conjecture states that if n divides the (n+1)th term of the fibonacci sequence, then n in prime.
# Tested 5,000,000,029 in 62.3 minutes

def fibonacci_prime_test(n):
    #This function uses the fibonacci test to test determine the input n's primality
    #First we generate the Fibonacci sequence mod n
    #No more than 4 entries are stored at a time for memory
    fib_seq = [0,1,1]
    i = 3
    while i <= n+1:
        next_element = (fib_seq[-1]+fib_seq[-2]) % n
        fib_seq.append(next_element)
        del fib_seq[0]
        i+=1

    #Then we test if the n+1th term or the n-1th of the sequence is divisible by n
    #If yes, n is prime. Otherwise, n is composite
    if (fib_seq[-1] == 0) or (fib_seq[0] == 0):
        return True
    return False

```

Python code

The following code was contributed by Nicholas Ruffolo and is contributed to Jamie Hayes. The Miller-Rabin test is a means to determine if an integer is a prime number or not, using probability similar to the Fermat test. This code can run itself 100,000,000 times to make it extremely probable that a number is prime in a few seconds for the largest prime in the range of 1-100,000,000. To make it very probable that a number is prime, we can have it run itself 40-60 times. For the smallest prime (2), the run time was 1.7 seconds. For the largest prime in our designated range (999999989), the run time was considerably larger, at 7 minutes and 12.5 seconds. The largest number I was able to find that could be ran in a reasonable amount of time was $(2^{19937}) - 1$ and it came out as true in 70 minutes. It should be noted, that for every prime that was tested here with this test, the result came out as true, but every prime with a run time here is a confirmed prime.

```

#v3

from random import randrange

def evaluate(n: int, t: int) -> bool:
    """
    n: number to be evaluated
    t: number of test iterations
    Function for running the Miller-Rabin primality test.
    Will run the test t times.
    Returns a boolean showing that n is either composite or probably prime.
    """

    if n == 2 or n == 3:
        return True

    if n > 2 and n % 2 == 0:
        return False  # n is even

    # we will halve m iteratively until we achieve the equality:
    # n - 1 = (2^k)m
    k = 0
    m = n - 1

    # this loop halves m
    while m % 2 == 0:
        k += 1
        m //= 2  # floor div

    # we now have k, m such that m is an odd integer

    for _ in range(t):
dcr
        a = randrange(2, n - 1)

        # getting initial value for b
        b = pow(a, m, n)

        # initial check for primality
        if b == 1 or b == n - 1:
            continue

        # iterate until a result is found
        for _ in range(k - 1):

            # raising b**2 and getting remainder from modulo n
            b = pow(b, 2, n)

            # checking value of b
            if b == n - 1:
```

```python
                break

        # if inner loop ends, check reason for ending
        # loop was broken
        if b == n - 1:
            continue
        # loop ran out
        else:
            return False

    # if all iterations were completed without throwing False, then n is probably prime
    return True
```

Python code

We also have some more code contributed by Nicholas Ruffolo on the Fermat primality test. Similar to the Miller-Rabin test, the Fermat Primality test uses probabilities as a method to see whether an integer is a prime number or not. Using this test, it was possible to verify the largest prime in our data range (99999989) in 8.8 seconds with 1 million runs of the test for verification. These two tests should be taken with a grain of salt, considering these tests only consider a number a prime though a high probability they are prime, rather than being deterministic/definitive in their testing. Written by Aanchal Tiwari

```python


#Fermat Primality Test

import random
# If n is prime, then always returns true,
# If n is composite than returns false with
# high probability Higher value of k increases
# probability of correct result
def isPrime(n, k):

  # Corner cases
  if n == 1 or n == 4:
    return False
  elif n == 2 or n == 3:
    return True

  # Try k times
  else:
    for i in range(k):

      # Pick a random number
      # in [2..n-2]
      # Above corner cases make
      # sure that n > 4
      a = random.randint(2, n - 2)

      # Fermat's little theorem
      if power(a, n - 1, n) != 1:
        return False

  return True

# Driver code
k = 3
if isPrime(11, k):
  print("true")
else:
  print("false")

if isPrime(15, k):
  print("true")
```

```
else:
  print("false")

# This code is contributed by Aanchal Tiwari
```

Python code

More on twin primes

The study of the twin prime conjecture is quite extensive. In fact Chen's theorem has achieved significant progress. There is also Brun's work on the sum of the reciprocals of the twin primes. It is believed that this sum converges and its value has been computed by using a significant range of prime numbers - much more comprehensive than our trivial range.

The following program was contributed by Nicholas Ruffolo and it estimates Brun's constant by using the primes in the trivial range. It was estimated to be 1.75881562. The run time for the code was 44.1 seconds. By considering the primes within the first 100 billion numbers one can find a better estimate, around 1.90216.

```python
#Brun's Constant
import pprint
import time # to time our program execution time
import pandas as pd # to create a dataframe and then convert to csv file

start_time = time.time()
def twinPrime1(n):

    # eratosthenes sieve - first we generate the primes using eratosthenes sieve,
    # where we set all values to true first, then set all non prime numbers as false
    prime = [True for i in range(n + 2)]
    p = 2
    while (p * p <= n + 1):
        if (prime[p] == True):
            for i in range(p * 2, n + 2, p):
                prime[i] = False
        p += 1

    #code for twin prime
    twin_prime1 = set()
    for p in range(2, n-1):
        if prime[p] and prime[p + 2]:
            twin_prime1.add((p))
    result = list(twin_prime1)
    result.sort()
    return(result)

    # Calling the function

import pprint
import time # to time our program execution time
import pandas as pd # to create a dataframe and then convert to csv file

start_time = time.time()
def twinPrime2(n):

    # eratosthenes sieve - first we generate the primes using eratosthenes sieve,
    # where we set all values to true first, then set all non prime numbers as false
    prime = [True for i in range(n + 2)]
    p = 2
    while (p * p <= n + 1):
        if (prime[p] == True):
            for i in range(p * 2, n + 2, p):
                prime[i] = False
        p += 1

    #code for twin prime
    twin_prime2 = set()
    for p in range(2, n-1):
        if prime[p] and prime[p + 2]:
            twin_prime2.add((p+2))
    result = list(twin_prime2)
    result.sort()
```

```
    return(result)

    # Calling the function

import PrimeB as B
import PrimeA as A
import pandas as pd
import summationcopy as sc
listb = B.twinPrime2(100000000)
lista = A.twinPrime1(100000000)
bruns = 0
for x,y in zip(lista,listb):
    bruns += (1/(x))+(1/(y))
print(bruns)
```

Python code

Prime numbers and graphics

Primes are in general very irregularly distributed. However there are some very striking attempts to visualize prime numbers within the natural numbers. One such visualization attempt is through the Ulam spiral.

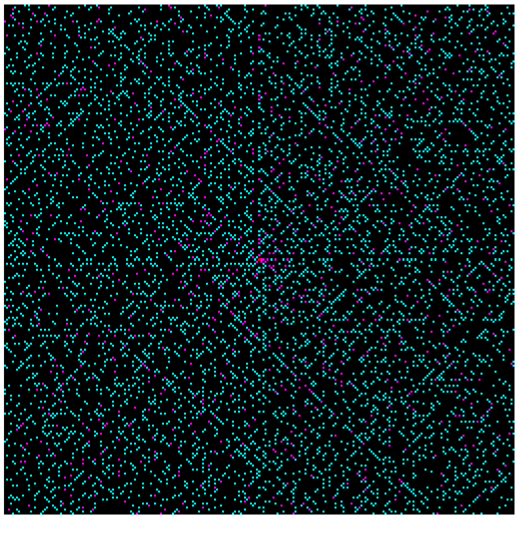
The following program was written by Vivek Ily and gives a visualization of the Ulam spiral.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

def make_spiral(arr):
    nrows, ncols= arr.shape
    idx = np.arange(nrows*ncols).reshape(nrows,ncols)[::-1]
    spiral_idx = []
    while idx.size:
        spiral_idx.append(idx[0])
        # Remove the first row (the one we've just appended to spiral).
        idx = idx[1:]
        # Rotate the rest of the array anticlockwise
        idx = idx.T[::-1]
    # Make a flat array of indices spiralling into the array.
    spiral_idx = np.hstack(spiral_idx)
    # Index into a flattened version of our target array with spiral indices.
    spiral = np.empty_like(arr)
    spiral.flat[spiral_idx] = arr.flat[::-1]
    return spiral
```

Python code to display the Ulam spiral

Here is an Ulam spiral visualization generated by Eddie Federmeyer. The general prime numbers are depicted in blue and the Germain primes in pink.



The following two programs were written by Nero Partida and they also implement the Ulam Spiral.

```

# Python code to print Ulam's spiral
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

# function to plot out the ulam spiral
```

```
def make_spiral(arr):
  nrows, ncols= arr.shape
  idx = np.arange(nrows*ncols).reshape(nrows,ncols)[::-1]
  spiral_idx = []
  while idx.size:
    spiral_idx.append(idx[0])

    # Remove the first row (the one we've
    # just appended to spiral).
    idx = idx[1:]

    # Rotate the rest of the array anticlockwise
    idx = idx.T[::-1]

  # Make a flat array of indices spiralling
  # into the array.
  spiral_idx = np.hstack(spiral_idx)

  # Index into a flattened version of our
  # target array with spiral indices.
  spiral = np.empty_like(arr)
  spiral.flat[spiral_idx] = arr.flat[::-1]
  return spiral

# edge size of the square array.
w = 251
# Prime numbers up to and including w**2.
primes = np.array([n for n in range(2,w**2+1) if all(
            (n % m) != 0 for m in range(2,int(np.sqrt(n))+1))])

# Create an array of boolean values: 1 for prime, 0 for composite
arr = np.zeros(w**2, dtype='u1')
arr[primes-1] = 1

# Spiral the values clockwise out from the centre
arr = make_spiral(arr.reshape((w,w)))

plt.matshow(arr, cmap=cm.binary)
plt.axis('off')
plt.show()
```

Python Code to generate an Ulam Spiral using primes.

```python

import numpy as np
import matplotlib.pyplot as plt

def primes_sieve_supercharged(n):
    sieve = np.ones(n // 2, dtype=np.bool)

    for i in range(3, int(n**0.5) + 1, 2):
        if sieve[i // 2]:
            sieve[i * i // 2::i] = False

    return np.concatenate(([2], 2 * np.nonzero(sieve)[0][1::] + 1))


def primes_sieve(n):
    # Array of possible primes
    primes = np.ones(n, dtype=bool)

    # 0 and 1 are not prime numbers
    primes[0] = False
    primes[1] = False

    for i in range(2, n):
        if primes[i]:
            primes[i**2::i] = False

    # Prime numbers are the indices of the True values
    return np.flatnonzero(primes)


def is_prime(n):
    if n <= 1:
        return False
    if n == 2:
        return True
    if n%2 == 0:
        return False

    for i in range(2, int(n**0.5)+1):
        if n%i == 0:
            return False

    return True


def primes_naive(n):
    primes = []

    for i in range(2, n):
        if is_prime(i):
            primes.append(i)

    return primes


def polar_plot(r, theta, area=0.01, show_grid=True):
    bg_color = '#000000'

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='polar')
    ax.set_yticklabels([])
    #ax.contour()
    #plt.set_cmap("")

```

```python
    if not show_grid:
        ax.grid(False)
        ax.set_facecolor(bg_color)
        fig.patch.set_facecolor(bg_color)

    ax.scatter(r, theta, marker="x", s=area)
    plt.show()

if __name__ == '__main__':
    num_primes = 100000

    p = primes_sieve_supercharged(num_primes)

    plots = [(num_primes, 0.05),(1000, 1.7)]

    # Plot it again, make it look nice
    for N, n in plots:
        polar_plot(p[:N], p[:N], area=n, show_grid=False)
```

Python Code to generate an Ulam Spiral using primes.