Background Concepts

Riemann Zeta of n

Rapidly converging series for zeta(n) for n odd were first discovered by Ramanujan (Zucker 1979, 1984, Berndt 1988, Bailey et al. 1997, Cohen 2000).

```
For n>1 and n=3 (mod 4),

 zeta(n)=(2^(n-1)pi^n)/((n+1)!)sum_(k=0)^((n+1)/2)(-1)^(k-1)(n+1; 2k)B_(n+1-2k)B_(2k)-2sum_(k=1)^inf
(92)
```

where B_k is again a Bernoulli number and (n; k) is a binomial coefficient. The values of the left-hand sums (divided by pi^n) in (92) for n=3, 7, 11, ... are 7/180, 19/56700, 1453/425675250, 13687/390769879500, 7708537/21438612514068750, ... (OEIS A057866 and A057867). For n>=5 and n=1 (mod 4)

```
import os
import mmap
import struct
import math
import pandas as pd

# Constants for block structure and file size
COLUMNS_PER_CHUNK = 64
CHUNKS_PER_PAGE = 64
PAGES_PER_BLOCK = 4096
BLOCKS_PER_FILE = 16

CHUNK_SIZE = COLUMNS_PER_CHUNK * 4  # Each integer is 32 bits (4 bytes)
PAGE_SIZE = CHUNK_SIZE * CHUNKS_PER_PAGE
BLOCK_SIZE = PAGE_SIZE * PAGES_PER_BLOCK
FILE_SIZE = BLOCK_SIZE * BLOCKS_PER_FILE  # Total size of the cache file

class PrimeCache:
    """
    PrimeCache handles storing and accessing prime numbers in a memory-mapped
file for efficient
    memory management. It lazily loads data into memory when needed, reducing
the memory footprint.
    """
    def __init__(self, filename):
        """
        Initializes the PrimeCache, creating or loading the memory-mapped
file.

        Arguments:
        filename -- the name of the binary file for caching prime numbers
```

```
        """
        self.filename = filename
        self.file = None
        self.mmap = None
        self.initialize_file()

    def initialize_file(self):
        """
        Creates the binary cache file if it does not exist, and memory-maps it
for efficient access.
        """
        if not os.path.exists(self.filename):
            with open(self.filename, 'wb') as f:
                f.write(b'\0' * FILE_SIZE)  # Initialize file with zeros
        self.file = open(self.filename, 'r+b')
        self.mmap = mmap.mmap(self.file.fileno(), 0)

    def close(self):
        """
        Closes the memory-mapped file and underlying file descriptor.
        """
        if self.mmap:
            self.mmap.close()
        if self.file:
            self.file.close()

    def get_offset(self, n):
        """
        Calculate the byte offset for the n-th value in the cache file.

        Arguments:
        n -- the index to retrieve the offset for

        Returns:
        offset -- the byte offset in the memory-mapped file
        """
        return (n // COLUMNS_PER_CHUNK) * CHUNK_SIZE + (n % COLUMNS_PER_CHUNK)
* 4

    def get_value(self, n):
        """
        Retrieves the value at index n in the cache file.

        Arguments:
        n -- the index to retrieve
```

```python
        Returns:
        value -- the value stored at index n
        """
        offset = self.get_offset(n)
        return struct.unpack('I', self.mmap[offset:offset+4])[0]

    def set_value(self, n, value):
        """
        Sets the value at index n in the cache file.

        Arguments:
        n -- the index to update
        value -- the value to set
        """
        offset = self.get_offset(n)
        self.mmap[offset:offset+4] = struct.pack('I', value)

    def get_is_prime(self, n):
        """
        Checks if the number at index n is marked as prime.

        Arguments:
        n -- the number to check

        Returns:
        True if prime, False otherwise
        """
        return self.get_value(n) != 0

    def set_is_prime(self, n, is_prime):
        """
        Marks the number at index n as prime or not prime.

        Arguments:
        n -- the index to update
        is_prime -- True if the number is prime, False otherwise
        """
        self.set_value(n, 1 if is_prime else 0)

def mod_check_with_dynamic_condition(n, cache):
    """
    Checks if a number is prime by performing modulus checks with known
primes.

    Arguments:
    n -- the number to check
```

```python
    cache -- the PrimeCache instance to retrieve prime information

    Returns:
    1 if the number is prime, 0 otherwise
    """
    for i in range(2, int(math.sqrt(n)) + 1):
        if cache.get_is_prime(i):
            if n % i == 0:
                return 0
    return 1

def generate_mod_table_dynamic(limit, cache):
    """
    Generates a table of prime numbers up to the given limit.

    Arguments:
    limit -- the upper limit to check for primes
    cache -- the PrimeCache instance to use for storing prime results

    Returns:
    table -- a list of tuples (n, prime_status) where prime_status is 1 if n
is prime, 0 otherwise
    """
    table = []
    max_cached_prime = 2

    # Find the largest cached prime
    while cache.get_value(max_cached_prime) != 0:
        max_cached_prime += 1
    max_cached_prime -= 1

    # Start checking from the next number after the largest cached prime
    for n in range(max_cached_prime + 1, limit + 1):
        if mod_check_with_dynamic_condition(n, cache):
            cache.set_is_prime(n, True)
            table.append([n, 1])
        else:
            cache.set_is_prime(n, False)
            table.append([n, 0])

    return table

def main():
    """
    Main function to prompt user input for the prime number limit, generate
the prime mod table,
```

```python
    and save the results in an Excel file.
    """
    cache = PrimeCache("prime_cache.bin")

    # Get user input for the limit
    limit = int(input("Enter the limit for generating prime numbers: "))

    print("Generating prime numbers...")

    # Generate the mod table
    mod_table_dynamic = generate_mod_table_dynamic(limit, cache)

    print("\nPrime Number Check Results:")
    for row in mod_table_dynamic:
        print(f"n = {row[0]}, prime = {row[1]}")

    # Save the results to an Excel file using pandas
    df = pd.DataFrame(mod_table_dynamic, columns=['Number', 'Prime Status'])
    df['Row Total'] = df['Prime Status']  # Row total: 1 for prime, 0
otherwise
    df.to_excel("prime_mod_table.xlsx", index=False)
    print("\nResults saved to prime_mod_table.xlsx")

    # Close the cache
    cache.close()

if __name__ == "__main__":
    main()
```

```python
def main_no_cache(limit):
    """
    Generates primes without using a cache and prints the prime numbers up to
the limit.
    """
    table = generate_mod_table_without_cache(limit)

    print("\nPrime Number Check Results (No Cache):")
    for row in table:
        print(f"n = {row[0]}, prime = {row[1]}")

    df = pd.DataFrame(table, columns=['Number', 'Prime Status'])
    df['Row Total'] = df['Prime Status']
    print(df)
```

```python
if __name__ == "__main__":
    limit = 101
    print("Generating primes without cache...")
    main_no_cache(limit)
```

```python
import unittest
import os
import pandas as pd
from io import StringIO

class TestPrimeCache(unittest.TestCase):
    def setUp(self):
        """
        Setup the cache file for testing.
        """
        self.cache_file = "test_prime_cache.bin"
        self.limit = 101
        if os.path.exists(self.cache_file):
            os.remove(self.cache_file)
        self.cache = PrimeCache(self.cache_file)

    def tearDown(self):
        """
        Close and remove the cache file after the test is done.
        """
        self.cache.close()
        if os.path.exists(self.cache_file):
            os.remove(self.cache_file)

    def test_prime_generation_with_cache(self):
        """
        Test prime number generation with the use of cache.
        """
        table = generate_mod_table_dynamic(self.limit, self.cache)
        primes = [row[0] for row in table if row[1] == 1]
        expected_primes = [
            2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
            61, 67, 71, 73, 79, 83, 89, 97
        ]
        self.assertEqual(primes, expected_primes)

    def test_prime_generation_without_cache(self):
        """
        Test prime number generation without cache, in-memory only.
        """
```

```
        table = generate_mod_table_without_cache(self.limit)
        primes = [row[0] for row in table if row[1] == 1]
        expected_primes = [
            2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
            61, 67, 71, 73, 79, 83, 89, 97
        ]
        self.assertEqual(primes, expected_primes)

def generate_mod_table_without_cache(limit):
    """
    Generates a table of prime numbers up to the given limit without using a
cache.

    Arguments:
    limit -- the upper limit to check for primes

    Returns:
    table -- a list of tuples (n, prime_status) where prime_status is 1 if n
is prime, 0 otherwise
    """
    table = []
    for n in range(2, limit + 1):
        is_prime = all(n % i != 0 for i in range(2, int(math.sqrt(n)) + 1))
        table.append([n, 1 if is_prime else 0])
    return table

if __name__ == "__main__":
    unittest.main()
```

Contemplation

If the infinite series of N is (0+1+2+...n) resulting in -1/12 Reimann Zeta...

And if the series of N for (0+1)=1; 1/2=0.5...

Then the 0.5 represents two possibilities:

1. When n is 1 then N is 0.5, and 0.5 represents 180°;
2. When n approaches infinity and is greater then 5, N is -1/12 representing 30°, as (Sine(0》 30°)=0.5, Cos(90》 60°)=0.5, Tan(30》 60°|60》 30°)=1;

Meaning: the Reimman Zeta is a Tangent of 45° of a complete series of 360°, where when larger then 5 forms 6 as a Radian turn of 1/2 of 1/12 of 2pi Radius. And where n>5 forms divergent curls of 30° in the Sine and Cosine with respect of the 45° intersection of 90°.

Mod(((Mod(n,12)+12)/3),4) = 1/3 of 1/12 as values 0..4 on 0.333... intervals from 0.00 to 3.66 where 4.00 is a 360° rotation of 0.33 back to 0.00.

The Reimann series of Mod(((Mod((n-sqrt(m),12)+12)/3),4) is -1/12 the Riemann Zeta of -1;

The Reimann series Mod(((Mod((m-sqrt(m),12)+12)/3),4) is 1/12 the Riemann Zeta of 1;

For limit of 0》4=0》4=0... 1/4th of 360° of the series N as 360° with respect to the n_th degree of N and the deviation of +/-1 as the m_th degree.

Background Concepts

Riemann Zeta of n

Rapidly converging series for zeta(n) for n odd were first discovered by Ramanujan (Zucker 1979, 1984, Berndt 1988, Bailey et al. 1997, Cohen 2000).

```
For n>1 and n=3 (mod 4),

 zeta(n)=(2^(n-1)pi^n)/((n+1)!)sum_(k=0)^((n+1)/2)(-1)^(k-1)(n+1; 2k)B_(n+1-2k)B_(2k)-2sum_(k=1)^inf
(92)
```

where B_k is again a Bernoulli number and (n; k) is a binomial coefficient. The values of the left-hand sums (divided by pi^n) in (92) for n=3, 7, 11, ... are 7/180, 19/56700, 1453/425675250, 13687/390769879500, 7708537/21438612514068750, ... (OEIS A057866 and A057867). For n>=5 and n=1 (mod 4)

```
import os
import mmap
import struct
import math
import pandas as pd

# Constants for block structure and file size
COLUMNS_PER_CHUNK = 64
CHUNKS_PER_PAGE = 64
PAGES_PER_BLOCK = 4096
BLOCKS_PER_FILE = 16

CHUNK_SIZE = COLUMNS_PER_CHUNK * 4  # Each integer is 32 bits (4 bytes)
PAGE_SIZE = CHUNK_SIZE * CHUNKS_PER_PAGE
BLOCK_SIZE = PAGE_SIZE * PAGES_PER_BLOCK
FILE_SIZE = BLOCK_SIZE * BLOCKS_PER_FILE  # Total size of the cache file

class PrimeCache:
    """
    PrimeCache handles storing and accessing prime numbers in a memory-mapped
file for efficient
    memory management. It lazily loads data into memory when needed, reducing
the memory footprint.
    """
    def __init__(self, filename):
        """
        Initializes the PrimeCache, creating or loading the memory-mapped
file.
```

```
        Arguments:
        filename -- the name of the binary file for caching prime numbers
        """
        self.filename = filename
        self.file = None
        self.mmap = None
        self.initialize_file()

    def initialize_file(self):
        """
        Creates the binary cache file if it does not exist, and memory-maps it
for efficient access.
        """
        if not os.path.exists(self.filename):
            with open(self.filename, 'wb') as f:
                f.write(b'\0' * FILE_SIZE)  # Initialize file with zeros
        self.file = open(self.filename, 'r+b')
        self.mmap = mmap.mmap(self.file.fileno(), 0)

    def close(self):
        """
        Closes the memory-mapped file and underlying file descriptor.
        """
        if self.mmap:
            self.mmap.close()
        if self.file:
            self.file.close()

    def get_offset(self, n):
        """
        Calculate the byte offset for the n-th value in the cache file.

        Arguments:
        n -- the index to retrieve the offset for

        Returns:
        offset -- the byte offset in the memory-mapped file
        """
        return (n // COLUMNS_PER_CHUNK) * CHUNK_SIZE + (n % COLUMNS_PER_CHUNK)
* 4

    def get_value(self, n):
        """
        Retrieves the value at index n in the cache file.
```

```
        Arguments:
        n -- the index to retrieve

        Returns:
        value -- the value stored at index n
        """
        offset = self.get_offset(n)
        return struct.unpack('I', self.mmap[offset:offset+4])[0]

    def set_value(self, n, value):
        """
        Sets the value at index n in the cache file.

        Arguments:
        n -- the index to update
        value -- the value to set
        """
        offset = self.get_offset(n)
        self.mmap[offset:offset+4] = struct.pack('I', value)

    def get_is_prime(self, n):
        """
        Checks if the number at index n is marked as prime.

        Arguments:
        n -- the number to check

        Returns:
        True if prime, False otherwise
        """
        return self.get_value(n) != 0

    def set_is_prime(self, n, is_prime):
        """
        Marks the number at index n as prime or not prime.

        Arguments:
        n -- the index to update
        is_prime -- True if the number is prime, False otherwise
        """
        self.set_value(n, 1 if is_prime else 0)

def mod_check_with_dynamic_condition(n, cache):
    """
    Checks if a number is prime by performing modulus checks with known
primes.
```

```
    Arguments:
    n -- the number to check
    cache -- the PrimeCache instance to retrieve prime information

    Returns:
    1 if the number is prime, 0 otherwise
    """
    for i in range(2, int(math.sqrt(n)) + 1):
        if cache.get_is_prime(i):
            if n % i == 0:
                return 0
    return 1

def generate_mod_table_dynamic(limit, cache):
    """
    Generates a table of prime numbers up to the given limit.

    Arguments:
    limit -- the upper limit to check for primes
    cache -- the PrimeCache instance to use for storing prime results

    Returns:
    table -- a list of tuples (n, prime_status) where prime_status is 1 if n
is prime, 0 otherwise
    """
    table = []
    max_cached_prime = 2

    # Find the largest cached prime
    while cache.get_value(max_cached_prime) != 0:
        max_cached_prime += 1
    max_cached_prime -= 1

    # Start checking from the next number after the largest cached prime
    for n in range(max_cached_prime + 1, limit + 1):
        if mod_check_with_dynamic_condition(n, cache):
            cache.set_is_prime(n, True)
            table.append([n, 1])
        else:
            cache.set_is_prime(n, False)
            table.append([n, 0])

    return table

def main():
```

```
    """
    Main function to prompt user input for the prime number limit, generate the prime mod table,
    and save the results in an Excel file.
    """
    cache = PrimeCache("prime_cache.bin")

    # Get user input for the limit
    limit = int(input("Enter the limit for generating prime numbers: "))

    print("Generating prime numbers...")

    # Generate the mod table
    mod_table_dynamic = generate_mod_table_dynamic(limit, cache)

    print("\nPrime Number Check Results:")
    for row in mod_table_dynamic:
        print(f"n = {row[0]}, prime = {row[1]}")

    # Save the results to an Excel file using pandas
    df = pd.DataFrame(mod_table_dynamic, columns=['Number', 'Prime Status'])
    df['Row Total'] = df['Prime Status']  # Row total: 1 for prime, 0 otherwise
    df.to_excel("prime_mod_table.xlsx", index=False)
    print("\nResults saved to prime_mod_table.xlsx")

    # Close the cache
    cache.close()

if __name__ == "__main__":
    main()
```

```
def main_no_cache(limit):
    """
    Generates primes without using a cache and prints the prime numbers up to the limit.
    """
    table = generate_mod_table_without_cache(limit)

    print("\nPrime Number Check Results (No Cache):")
    for row in table:
        print(f"n = {row[0]}, prime = {row[1]}")

    df = pd.DataFrame(table, columns=['Number', 'Prime Status'])
```

```
    df['Row Total'] = df['Prime Status']
    print(df)

if __name__ == "__main__":
    limit = 101
    print("Generating primes without cache...")
    main_no_cache(limit)
```

```
import unittest
import os
import pandas as pd
from io import StringIO

class TestPrimeCache(unittest.TestCase):
    def setUp(self):
        """
        Setup the cache file for testing.
        """
        self.cache_file = "test_prime_cache.bin"
        self.limit = 101
        if os.path.exists(self.cache_file):
            os.remove(self.cache_file)
        self.cache = PrimeCache(self.cache_file)

    def tearDown(self):
        """
        Close and remove the cache file after the test is done.
        """
        self.cache.close()
        if os.path.exists(self.cache_file):
            os.remove(self.cache_file)

    def test_prime_generation_with_cache(self):
        """
        Test prime number generation with the use of cache.
        """
        table = generate_mod_table_dynamic(self.limit, self.cache)
        primes = [row[0] for row in table if row[1] == 1]
        expected_primes = [
            2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
            61, 67, 71, 73, 79, 83, 89, 97
        ]
        self.assertEqual(primes, expected_primes)

    def test_prime_generation_without_cache(self):
```

```
        """
        Test prime number generation without cache, in-memory only.
        """
        table = generate_mod_table_without_cache(self.limit)
        primes = [row[0] for row in table if row[1] == 1]
        expected_primes = [
            2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
            61, 67, 71, 73, 79, 83, 89, 97
        ]
        self.assertEqual(primes, expected_primes)

def generate_mod_table_without_cache(limit):
    """
    Generates a table of prime numbers up to the given limit without using a
cache.

    Arguments:
    limit -- the upper limit to check for primes

    Returns:
    table -- a list of tuples (n, prime_status) where prime_status is 1 if n
is prime, 0 otherwise
    """
    table = []
    for n in range(2, limit + 1):
        is_prime = all(n % i != 0 for i in range(2, int(math.sqrt(n)) + 1))
        table.append([n, 1 if is_prime else 0])
    return table

if __name__ == "__main__":
    unittest.main()
```

Contemplation

If the infinite series of N is (0+1+2+...n) resulting in -1/12 Reimann Zeta...

And if the series of N for (0+1)=1; 1/2=0.5...

Then the 0.5 represents two possibilities:

1. When n is 1 then N is 0.5, and 0.5 represents 180°;
2. When n approaches infinity and is greater then 5, N is -1/12 representing 30°, as (Sine(0》30°)=0.5, Cos(90》60°)=0.5, Tan(30》60°|60》30°)=1;

Meaning: the Reimman Zeta is a Tangent of 45° of a complete series of 360°, where when larger then 5 forms 6 as a Radian turn of 1/2 of 1/12 of 2pi Radius. And where n>5 forms divergent curls of 30° in the Sine and Cosine with respect of the 45° intersection of 90°.

Mod(((Mod(n,12)+12)/3),4) = 1/3 of 1/12 as values 0..4 on 0.333... intervals from 0.00 to 3.66 where 4.00 is a 360° rotation of 0.33 back to 0.00.

The Reimann series of Mod(((Mod((n-sqrt(m),12)+12)/3),4) is -1/12 the Riemann Zeta of -1;

The Reimann series Mod(((Mod((m-sqrt(m),12)+12)/3),4) is 1/12 the Riemann Zeta of 1;

For limit of 0》4=0》4=0... 1/4th of 360° of the series N as 360° with respect to the n_th degree of N and the deviation of +/-1 as the m_th degree.

\Delta_{\text{Riemann}} = \nabla^2 f - \left(\frac{\partial m(y)}{\partial x} - \frac{\partial n(x)}{\partial y} \right)

Now with taking n-sqrt(m) or m-sqrt(n) as A arbitrary label.

Mod(A,12)/3 =B Mod(B,4)=[0..4] in intervals as [0.00, 0.33, 0.66, 1.00, 1.33, 1.66, 2.00, 2.33, 2.66, 3.00, 3.33, 3.66;] then 4.00=0.00 back to beginning.

If the number begins with 0 it's first 90°, then 90 to 180 etc as 1, 2, and 3.

The decimals .00, .33, and .66 mean tan, Sine, and cos respectively.

Value as difference in curl/vector space as limit L as -1.0 to 1.0 and tan(L[A]) etc is going to result in:

1. When taking Sine it is limit 0° to 30° as 0.00 approaching 0.33;

2. When taking Cosine it is limit 90° to 60° as 1.00 approaching 0.66;

3. When taking Tangent it is limit 30° to 45°, or 60° to 45°, as either:

- 0.33 approaching 0.66,
- 0.66 approaching 0.33,
- 0.66 approaching 1.00,
- 0.33 approaching 0.00, For the Tangent squared, or as needing 1/2 or a square root trigonometrically of it.

Meaning, each tan(45) is actually half of the 90° quadrant, and the 1/3rd of the 90° as 30° is the remainder in 15° segments as approximately (0.333... + 0.666...)/2=(0.999)/2=(0.45454535)

Is N and M are cycles of 1..12 back to 1, and it represents proportional segments of 360° curl deviation of the Vector field, then the Reimmen Theta difference is the 15° Tangent as the intersects between 0.33... and 0.66... to 0.00, 0.50, and 1.00 absolute of Tangent(45), Sine(30), or Cosine(60), where 0.00 or 1.00 are both equal to a multiple of 1.00 from 0 thru 3 `[0,1,2,3》0,1,2,3》...0,1,2,3...inf]` out of 4 quadrants of 90° in the 360° circle of the complete vector space as 2×180° Bidirectional difference in Curl with respects to the Laplacian difference of the Divergent and Gradient as 180° vectors relative to their 180° perpendicular plane.

---

1. Take all N and M for `1,2,3,4,5,6,7 8 9 10,11,12` as 12×12 matrix.
2. For each N and M intersect determine:
    1. A1=N-M
    2. A2=M-N
    3. A3=N-(sqrt(M))
    4. A4=M-(sqrt(N))
    - B=Mod(A,12)/3
    - C=Mod(B,4)
3. For A1 and A2:
    1. B represents the deviation of -1/12 as the Riemann Series for (-1);
    2. C represents the direction of the Curl for the N and M with respect to the total series of n in N and m in M as 90° rotations:
        - 0.00,0.33,066;

        - 1.00,1.33,1.66;
        - 2.00,2.33,2.66;
        - 3.00,3.33,3.66;
        - 4.00=0.00=360°/4=90°
        - 90°/3=30°
    3. When difference is:
        - Q.00, Tangent of 45° = 1.00
        - Q.33, Sine of 30° = 0.50
        - Q.66, Cosine of 60° =0.50
4. For A3 and A4:
    1. B represents the deviation of 360°/2=180° for 2 180° vectors being compared;
    2. C represents the deviation from 90° to 45° to form the Tangent of 45° in A1/A2;
5. Then, the deviation of 1/8th of 360° as 2×1/8=1/4 of 360° as 90° quadrants is the -1/12th as 30° segments:
    1. Sin when 00° to 30°;
    2. Tan when 30° to 60°;
    3. Cos when 60° to 90°;
    4. As Sin=0.5, Tan=1.0, Cos=0.5;
6. As the infinite series of n grows to N with respect to m growing to M, and where M is N-1 as m is n-1 arrays from `[1..12]` or products of the 1/12th to form the Riemann Zeta of -1 equals -1/12; the deviation of angles in respect to indexes is in 1/12th of 360° as 2×180° infinite vectors deviation in curl along the series.