```python

from random import randrange
import numpy as np


########## 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)

```

```python
      # 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


########## 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.

For this sieve to function, we implement a sequence of smaller algorithms to parse out primes.

First, we remove any numbers on the interval that are multiples of smaller trivial primes. Prime Number Theorem estimates that the prime numbers found up to 10000 represent approximately 80% (tbd idk what the true value is) of all possible primes. As such, removing these obvious composite numbers reduces the the number of numbers to test significantly.

We then push the remaining numbers through the Miller-Rabin test. This primaility test is probabalistic, but is extremely fast and has an error rate of approximately 2^{-128}. This will effectively remove any composite numbers left.

Lastly, using the Fibonacci test, we identify deterministically which of the remaining numbers are true primes.

The High Sieve algorithm was built in order to more easily parse true prime numbers out of the non-trivial range, which is defined to be $[0, 10^{10}]$. The objective in doing so was to compare the quantities of primes found in the non-trivial range to the estimations provided by Prime Number Theorem.

The Prime Counting Function, $\pi(x)$, can be used to estimate the quantity of primes not greater than x using the formula $\frac{x}{\log x}$.

We can then alter this formula to estimate the quantity of primes on an interval [x, x+k].

$$\frac{x+k}{log(x+k)} - \frac{x}{log(x+k)} \to \frac{k}{log(x+k)}$$

To test the estimates of $\pi(x)$ against this sieve, we looked at increasingly high intervals outside of the trivial range and compared the true number of primes to $\pi(x)$.

During testing, we found that the difference between $\pi(x)$ and the true prime count approaches zero as the lower bound for the interval tends to infinity.