**Algorithm Study Template**

**Algorithm**: Sieve of Atkin

**aka**: <no alternative names>

**Techniques**: Sieving, Wheel Factorization, Modular Arithmetic

**Categories**: Searching, Prime Number Sieving

**Problem**: The Sieve of Atkin is a faster, more modern way of finding all prime numbers up to a specified integer. It is an optimized version of the Sieve of Eratosthenes that includes a considerable amount of preliminary work prior to marking off multiples of the square of each prime. By contrast, the Sieve of Eratosthenes only marks off the multiples of primes with no preliminary work.

**Applications**: Algorithms that generate prime numbers are used in various applications, such as hashing, public-key cryptography, and searching for prime factors in large numbers. A more specific example would be using the Sieve of Atkin to find that there are 168 prime numbers less than or equal to 1000.

**References**:

* <http://www.ams.org/journals/mcom/2004-73-246/S0025-5718-03-01501-1/S0025-5718-03-01501-1.pdf>
* <http://en.wikipedia.org/wiki/Wheel_factorization>
* <http://math.stackexchange.com/questions/59041/understanding-the-sieve-of-atkin>
* <http://stackoverflow.com/questions/10580159/sieve-of-atkin-explanation-and-java-example>
* <http://en.wikipedia.org/wiki/Talk:Sieve_of_Atkin#How_is_this_faster_than_the_Sieve_of_Eratosthenes.3F>

**Implementation details**:

* **Big Idea**: The Sieve of Atkin uses three quadratic equations and an additional check for perfect squares and their multiples to find all of the prime integers less than a given integer limit.
* **Description**:

First, a boolean array with limit+1 elements is declared and set to false. We use limit+1 because we start indexing from one instead of zero. Then, we set the numbers two and three to be primes by default. This is because the algorithm starts checking from five. We don’t have to worry about limits of three or four because, although checking starts at five, the two primes to be found have already been set. The drawback of setting these two primes ahead of time is that limits below three will cause an array out-of-bounds error because the code will still try and set three to true. This is an edge case that I chose to handle by disallowing limits less than three when the user inputs their chosen limit.

Now we step into the real meat of the algorithm. Using wheel factorization and modular arithmetic, we determine whether there are an odd number of (x,y) pairs (x and y being individually greater than or equal to one and less than or equal to the square root of the limit) that solve three different quadratic equations. This is done by first initializing an integer n to n=(4\*x^2)+(y^2). Then we check that n is less than or equal to the limit and that n mod twelve is equal to either one or five. If the check passes, then array[n] is set to true, indicating that n is a prime number.

Next we set n to n=(3\*x^2)+(y^2) and check that n is less than or equal to the limit and that n mod twelve is equal to seven. If the check passes, then array[n] is set to true, indicating that n is a prime number. After that, we set n to n=(3\*x^2)-(y^2) and check that x is greater than y, n is less than or equal to the limit, and n mod twelve is equal to eleven. If the check passes, then array[n] is set to true, indicating that n is a prime number.

The final step of the algorithm is to iterate through the array, starting from five, and ensure that none of the numbers currently marked prime are perfect squares. Perfect squares, some of which may not have been found before now, and all of their multiples are now marked as composite. Following this last step, the primes can be printed out or counted and reported as a total number of primes.

* **Pseudo-code**: (adapted from Wikipedia)

// set arbitrary search limit

limit ← 1000000

// initialize the sieve by creating a boolean array set to false

boolean array [limit + 1]

for i in [0, limit + 1]: prime(i) ← false

//since algorithm starts at 5, 2 and 3 are set to true by default

array[2] = true

array[3] = true

// put in candidate primes:

// integers which have an odd number of

// representations by certain quadratic forms

for (x, y) in [1, √limit] × [1, √limit]:

n ← 4x²+y²

if (n ≤ limit) and (n mod 12 = 1 or n mod 12 = 5):

is\_prime(n) ← !is\_prime(n)

n ← 3x²+y²

if (n ≤ limit) and (n mod 12 = 7):

is\_prime(n) ← !is\_prime(n)

n ← 3x²-y²

if (x > y) and (n ≤ limit) and (n mod 12 = 11):

is\_prime(n) ← !is\_prime(n)

// eliminate perfect square composites by sieving

for n in [5, √limit]:

if is\_prime(n):

// n is prime, omit multiples of its square; this is

// sufficient because composites which managed to get

// on the list cannot be square-free

is\_prime(k) ← false, k ∈ {n², 2n², 3n², ..., limit}

//report the primes found

for n in [2, limit]:

if is\_prime(n): print n

* **Specific implementation**: (see Atkin.java)

**Correctness**:

**Theoretical**: Complete proofs are provided in the algorithm authors’ original paper, located at: <http://www.ams.org/journals/mcom/2004-73-246/S0025-5718-03-01501-1/S0025-5718-03-01501-1.pdf>

**Empirical**: For testing purposes, this program allows the user to enter an integer limit greater than two. As previously stated, limits below three will cause an error during the program’s execution. The input is verified using a try-catch statement, as well as an if-statement. If unacceptable input if found, the user is notified and the program terminates.

Once an acceptable limit has been input, the program runs the Sieve of Atkin to find the number of primes less than or equal to the limit. For comparison purposes, the Sieve of Eratosthenes is also run on the same limit for the same purpose. Since I already had code to run the other sieve, I thought it would be a wasted opportunity not to do so. The resulting arrays of primes for each method are iterated through by a primality test to ensure that all found primes are legitimate. Then, the numbers of primes found by both tests are compared for equality. If a false prime is found or if the two methods output differing results, the user is informed of an error and the program terminates. With that in mind, the following test results are intended to show that the methods for finding primes are correct and that they both returned the same results in all cases.

Test 1:

Prime numbers <= 500,000,000 (five hundred million)

Sieve of Eratosthenes:

There are 26355867 prime numbers <= 500000000

Sieve of Atkin:

There are 26355867 prime numbers <= 500000000

Test 2:

Prime numbers <= 300,000,000 (three hundred million)

Sieve of Eratosthenes:

There are 16252325 prime numbers <= 300000000

Sieve of Atkin:

There are 16252325 prime numbers <= 300000000

Test 3:

Prime numbers <= 250,000,000 (two hundred and fifty million)

Sieve of Eratosthenes:

There are 13679318 prime numbers <= 250000000

Sieve of Atkin:

There are 13679318 prime numbers <= 250000000

Test 4:

Prime numbers <= 200,000,000 (two hundred million)

Sieve of Eratosthenes:

There are 11078937 prime numbers <= 200000000

Sieve of Atkin:

There are 11078937 prime numbers <= 200000000

Test 5:

Prime numbers <= 100,000,000 (one hundred million)

Sieve of Eratosthenes:

There are 5761455 prime numbers <= 100000000

Sieve of Atkin:

There are 5761455 prime numbers <= 100000000

Test 6:

Prime numbers <= 85,000,000 (eighty-five million)

Sieve of Eratosthenes:

There are 4943731 prime numbers <= 85000000

Sieve of Atkin:

There are 4943731 prime numbers <= 85000000

Test 7:

Prime numbers <= 75,000,000 (seventy-five million)

Sieve of Eratosthenes:

There are 38703181 prime numbers <= 750000000

Sieve of Atkin:

There are 38703181 prime numbers <= 750000000

**Performance**:

**Theoretical**: Via Wikipedia: (The Sieve of Atkin) computes primes up to *N* using O(*N*/log log *N*) operations with only *N*1/2 + o(1) bits of memory. That is a little better than the sieve of Eratosthenes which uses O(*N*) operations and O(*N*1/2(log log *N*)/log *N*) bits of memory.

**Empirical**: The following execution times are only intended to measure the amount of time taken to identify all primes less than or equal to the limit and count up the total number of primes found. No printed output or other “noise” is intentionally included in the measurements.

Test 1:

Prime numbers <= 500,000,000 (five hundred million)

The Sieve of Eratosthenes was completed in 10026.341 milliseconds.

The Sieve of Atkin was completed in 12086.126 milliseconds.

Test 2:

Prime numbers <= 300,000,000 (three hundred million)

The Sieve of Eratosthenes was completed in 7441.318 milliseconds.

The Sieve of Atkin was completed in 8197.19 milliseconds.

Test 3:

Prime numbers <= 250,000,000 (two hundred and fifty million)

The Sieve of Eratosthenes was completed in 8061.955 milliseconds.

The Sieve of Atkin was completed in 7685.113 milliseconds.

Test 4:

Prime numbers <= 200,000,000 (two hundred million)

The Sieve of Eratosthenes was completed in 6165.665 milliseconds.

The Sieve of Atkin was completed in 5761.146 milliseconds.

Test 5:

Prime numbers <= 100,000,000 (one hundred million)

The Sieve of Eratosthenes was completed in 3207.831 milliseconds.

The Sieve of Atkin was completed in 2398.365 milliseconds.

Test 6:

Prime numbers <= 85,000,000 (eighty-five million)

The Sieve of Eratosthenes was completed in 2849.653 milliseconds.

The Sieve of Atkin was completed in 2170.25 milliseconds.

Test 7:

Prime numbers <= 75,000,000 (seventy-five million)

The Sieve of Eratosthenes was completed in 13810.184 milliseconds.

The Sieve of Atkin was completed in 16199.577 milliseconds.

(Note that the times for these performance tests correspond to the same tests detailed in the “Empirical Correctness” section of this report. That is why I have specifically given a number to each test.)

I chose to report these specific tests for a reason. I believe that this set of test data shows a bound or “sweet spot”, so to speak, where the Sieve of Atkin performs better by executing faster than the Sieve of Eratosthenes. The Sieve of Atkin begins being faster around the time we reach datasets of eighty-five million and stops performing better when we reach datasets larger than around two hundred and fifty million. I tested other datasets larger and smaller than these I’m reporting as well, and from what I see, these are the apparent bounds where my implementation performs better. I happened upon a talk page on Wikipedia where some contributors were discussing which sieve is faster, and the consensus seemed to be that the performance of one over the other is extremely sensitive to the specific optimizations of the implementation. So as far as whether or not the Sieve of Atkin performs better than the Sieve of Eratosthenes, my official answer is “It can, but it doesn’t always.”

**Anecdotes**: As an aside, I debated back and forth about whether or not to work on another prime number sieve, but the Sieve of Atkin’s logic was different from the Sieve of Eratosthenes in a way that made me want to test them out and see which is superior in terms of performance.

**History**: The Sieve of Atkin was written by A. O. L. Atkin and Daniel J. Bernstein in 2004. It is recognized as an optimized version of the Sieve of Eratosthenes that performs better in most cases, depending upon the optimizations of the implementation.

**Variations**: The Sieve of Atkin is itself a variation of the Sieve of Eratosthenes. I am unable to find any direct variations of the Sieve of Atkin.

**Alternatives**: The Sieve of Eratosthenes and the Sieve of Sundaram are each acceptable for the same tasks as the Sieve of Atkin. They take an integer limit and find all prime numbers less than or equal to that limit. The Sieve of Eratosthenes uses the much simpler method of crossing out multiples of primes. The Sieve of Sundaram is very similar to the Sieve of Eratosthenes, but does not consider even numbers, choosing to mark them all as composite by crossing out multiples of two as an additional step to the algorithm.

**Credits:**

* <http://en.wikipedia.org/wiki/Sieve_of_Atkin>