Fastest Algorithm to Find Prime Numbers

Last modified: October 27, 2021

by Akbar Karimi

Math and Logic

1. Overview

Prime numbers have always been an interesting topic to dive into. However, no one has been able to find a clean and finite formula to generate them. Therefore, mathematicians have relied on algorithms and computational power to do that. Some of these algorithms can be time-consuming while others can be faster.

In this tutorial, we'll go over some of the well-known algorithms to find prime numbers. We'll start with the most ancient one and end with the most recent one.

Most algorithms for finding prime numbers use a method called prime sieves. Generating prime numbers is different from determining if a given number is a prime or not. For that, we can use a primality test such as Fermat primality test or Miller-Rabin method. Here, we only focus on algorithms that find or enumerate prime numbers.

2. Sieve of Eratosthenes

Sieve of Eratosthenes is one of the oldest and easiest methods for finding prime numbers up to a given number. It is based on marking as composite all the multiples of a prime. To do so, it starts with 2 as the first prime number and marks all of its multiples (`4, 6, 8, ...`). Then, it marks the next unmarked number (3) as prime and crosses out all its multiples (`6, 9, 12, ...`). It does the same for all the other numbers up to n:

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

However, as we can see, some numbers get crossed several times. In order to avoid it, for each prime p, we can start from p^2 to mark off its multiples. The reason is that once we get to a prime p in the process, all its multiples smaller than p^2 have already been crossed out. For example, let's imagine that we get to 7. Then, we can see that `14, 21, 28, 35,` and `42` have already been marked off by `2, 3, 2, 5,` and 3. As a result, we can begin with `49`.

We can write the algorithm in the form of pseudocode as follows:

```
Algorithm 1: Algorithm for the sieve of Eratosthenes
Data: An arbitrary number (n)
Result: Prime numbers smaller than n
Function findPrimes(n)
    A ← an array of size n with boolean values set to True
    for i ← 2 to √n do
        if A[i] is True then
            j ← i^2
            while j ≤ n do
                A[j] ← False
                j ← j + i
            end
        end
    end
    return Indices of A elements that are True
end
```

In order to calculate the complexity of this algorithm, we should consider the outer `for` loop and the inner `while` loop. It's easy to see that the former's complexity is $O(\sqrt{n})$. However, the latter is a little tricky. Since we enter the `while` loop when i is prime, we'll repeat the inner operation $\frac{\sqrt{n}}{p}$ number of times, with p being the current prime number. As a result, we'll have:

$$\sqrt{n}\sum_{p<\sqrt{n}}\frac{\sqrt{n}}{p} = n\sum_{p<\sqrt{n}}\frac{1}{p}$$

In their book (theory of numbers), Hardy and Wright show that $\sum_{p<n}\frac{1}{p} = \log\log n + O(1)$. Therefore, the time complexity of the sieve of Eratosthenes will be $O(n\log\log n)$.

3. Sieve of Sundaram

This method follows the same operation of crossing out the composite numbers as the sieve of Eratosthenes. However, it does that with a different formula. Given i and j less than n, first we cross out all the numbers of the form $i + j + 2ij$ less than n. After that, we double the remaining numbers and add 1. This will give us all the prime numbers less than $2n + 1$. However, it won't produce the only even prime number (2).

Here's the pseudocode for this algorithm:

```
Algorithm 2: Algorithm for the sieve of Sundaram
Data: An arbitrary number (n)
Result: Prime numbers smaller than n
Function findPrimes(n)
    k ← ⌊(n−1)/2⌋
    A ← an array of size k + 1 with boolean values set to True
    for i ← 1 to √k do
        j ← i
        while i + j + 2 × i × j ≤ k do
            A[i + j + 2 × i × j] ← False
            j ← j + 1
        end
    end
    T ← Indices of A elements that are True
    T ← 2 × T + 1
    return T
end
```

We should keep in mind that with n as input, the output is the primes up to $2n + 1$. So, we divide the input by half, in the beginning, to get the primes up to n.

We can calculate the complexity of this algorithm by considering the outer `for` loop, which runs for $\sqrt{n}$ times, and the inner `while` loop, which runs for less than $\frac{\sqrt{n}}{i}$ times. Therefore, we'll have:

$$\sqrt{n}\sum_{i<\sqrt{n}}\frac{\sqrt{n}}{i} = n\sum_{i<\sqrt{n}}\frac{1}{i}$$

This looks like a lot similar to the complexity we had for the sieve of Eratosthenes. However, there's a difference in the values i can take compared to the values of p in the sieve of Eratosthenes. While p could take only the prime numbers, i can take all the numbers between 1 and $\sqrt{n}$. As a result, we'll have a larger sum. Using the direct comparison test for this harmonic series, we can conclude that:

$$\sum_{i=1}^{i=\sqrt{n}}\frac{1}{i} \geq 1 + \frac{\log n}{4}$$

As a result, the time complexity for this algorithm will be $O(n\log n)$.

4. Sieve of Atkin

Sieve of Atkin speeds up (asymptotically) the process of generating prime numbers. However, it is more complicated than the others.

First, the algorithm creates a sieve of prime numbers smaller than 60 except for `2, 3, 5`. Then, it divides the sieve into 3 separate subsets. After that, using each subset, it marks off the numbers that are solutions to some particular quadratic equation and that have the same modulo-sixty remainder as that particular subset. In the end, it eliminates the multiples of square numbers and returns `2, 3, 5` along with the remaining ones. The result is the set of prime numbers smaller than n.

We can express the process of the sieve of Atkin using pseudocode:

```
Algorithm 3: Algorithm for the sieve of Atkin
Data: An arbitrary number (n)
Result: Prime numbers smaller than n
Function findPrimes(n)
    S ← {1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59}
    A ← an array of size n with boolean values set to False
    for x ← 1 to √n do
        for y ← 1 to √n by 2 do
            m ← 4x^2 + y^2
            if m ≡ {1, 13, 17, 29, 37, 41, 49, 53}  mod 60 and m <= n
             then
                A[m] ← ¬A[m]
        end
    end
    for x ← 1 to √n by 2 do
        for y ← 2 to √n by 2 do
            m ← 3x^2 + y^2
            if m ≡ {7, 19, 31, 43}  mod 60 and m <= n then
                A[m] ← ¬A[m]
        end
    end
    for x ← 2 to √n do
        for y ← x − 1 to 1 by −2 do
            m ← 3x^2 − y^2
            if m ≡ {11, 23, 47, 59}  mod 60 and m <= n then
                A[m] ← ¬A[m]
        end
    end
    M ← 60 × w + s where w ∈ {0, 1, 2, ..., n/60} and s ∈ S
    for m in M − {1} do
        if m^2 > n then
            Break;
        else
            mm ← m^2
            if A[m] is True then
                for m2 in M do
                    c ← mm × m2
                    if c > n then
                        Break;
                    else
                        A[c] ← False
                end
    end
    primes ← {2, 3, 5}
    primes.append({True elements of A})
    return primes
end
```

It is easy to see that the first three `for` loops in the sieve of Atkin require $O(n)$ operations. To conclude that the last loop also runs in $O(n)$ time, we should pay attention to the condition that will end the loop. Since when m^2 and c, which is a multiple of a square number, are greater than n, we get out of the loops, they both run in $O(\sqrt{n})$ time. As a result, the asymptotic running time for this algorithm is $O(n)$.

Comparing this running time with the previous ones shows that the sieve of Atkin is the fastest algorithm for generating prime numbers. However, as we mentioned, it requires a more complicated implementation. In addition, due to its complexity, this might not even be the fastest algorithm when our input number is small but if we consider the asymptotic running time, it is faster than others.

5. Conclusion

In this article, we reviewed some of the fast algorithms that we can use to generate prime numbers up to a given number.

If you have a few years of experience in Computer Science or research, and you're interested in sharing that experience with the community, have a look at our **Contribution Guidelines**.