A Method for Generating Prime Numbers
(A Constructive Approach)

Felipe A. Wescoup, PE

2320 Point Star Drive

Arlington TX 76001

Abstract

This paper presents a method for generating lists of prime numbers. The algorithm presented calculates the set of all composite (non-prime) numbers up to a given limit and the set of primes is subsequently defined as its complement. The method is demonstrated through a JavaScript implementation, which is evolved over four iterative levels increasing optimization. Level 1 presents the core mathematical concept. Levels 2 and 3 include logical assumptions that improve efficiency. Level 4 is capable of calculating all primes up to 20 million in seconds within a standard web browser. This paper is supplemented by a public GitHub repository containing the complete operational code.

1.0 Introduction: The Sieve and the Set

This section discusses the two typical methods for finding prime numbers which are trial division and a sieve-based approach. This section also introduces a constructive method for generating composite numbers.

1.1 Trial Division

The foundational method for identifying primes is **trial division**. To determine if a number, such as 91, is prime, trial division attempts division by smaller integers. The process involves checking potential divisors sequentially. If any number is found to divide 91 without a remainder, it is proven to be composite. For the number 91, this process reveals that 7 is a divisor (91 / 7 = 13), thus proving that 91 is composite.

This method can be refined; it is only necessary to test for divisibility by prime numbers up to the square root of the target number. To test if 97 is prime, for instance, the only prime divisors that need to be checked are 2, 3, 5, and 7, as the square root of 97 is approximately 9.8. As none of these primes divide 97 evenly, it can be concluded that 97 is prime.

This traditional approach is fundamentally a test of *inclusion*, directly testing each integer for membership in the set of primes. While effective for smaller numbers, this becomes incredibly inefficient as the magnitude of the integers under consideration increases.

1.2 A Sieve-Based Approach

An alternative to the inclusion-based method of trial division is a process of *exclusion*. Rather than testing individual numbers for primality, this approach systematically eliminates composite numbers from a list of integers. The most famous example of this technique is the **Sieve of Eratosthenes**.

The algorithm begins with a complete list of integers from 2 up to a specified limit. The first number in the list, 2, is prime. All multiples of 2 (4, 6, 8, and so on) are then marked as composite and eliminated. The process then moves to the next number that has not been eliminated, which is 3. All multiples of 3 are subsequently marked as composite. This procedure is repeated, with each successive non-eliminated number being identified as prime and its multiples removed. The numbers that remain at the end of this process are the prime numbers within the specified range.

This sieve method offers a significant performance advantage over trial division when the goal is to find all prime numbers up to a given limit, as it avoids redundant checks by eliminating entire classes of composite numbers in single passes.

1.3 A Constructive Method for Generating Composites

This paper presents a third alternative, a *constructive* approach, distinct from both inclusion and exclusion. Rather than testing integers one-by-one or filtering out non-primes (as a sieve does), this method directly generates the complete set of composite numbers through a deterministic function.

The core of this approach lies in a single function that, when evaluated over a specified range, produces every composite number and no prime numbers. This shifts the problem from one of identification or elimination to one of direct construction. By defining the mathematical structure of non-primes, the set of primes is subsequently defined as its complement, the set of integers not generated by the function.

2.0 The Core Algorithm: Generating Non-Primes

The algorithm functions by maintaining two sets: prime (the list of primes found) and nonprime (the list of composite numbers to be excluded).

The process begins by handling the only even prime number, 2. The number 2 is added to the prime set, and the algorithm's main loop is configured to check only odd numbers from 3 upward. This immediately halves the number of candidates to check.

The logic proceeds as a step-by-step loop:

1. **Start at x = 3.**
   - The prime set is [2].
   - The nonprime set is empty.
2. Is 3 in nonprime? No.
   - Add 3 to the prime set. The prime set is now [2, 3].
   - Generate composite numbers to add to nonprime. The algorithm multiplies 3 by all *odd* primes in the prime set.
   - 3 * 3 = 9. Add 9 to nonprime.
   - The nonprime set is now [9].
3. **Increment to x = 5.**
4. Is 5 in nonprime? No.
   - Add 5 to the prime set. The prime set is now [2, 3, 5].
   - Generate composite numbers by multiplying 5 by all odd primes found so far ([3, 5]).
   - 5 * 3 = 15. Add 15 to nonprime.
   - 5 * 5 = 25. Add 25 to nonprime.
   - The nonprime set is now [9, 15, 25].
5. **Increment to x = 7.**
6. Is 7 in nonprime? No.
   - Add 7 to the prime set. The prime set is now [2, 3, 5, 7].
   - Generate composite numbers (7 * 3, 7 * 5, 7 * 7).
   - Add 21, 35, and 49 to nonprime.
   - The nonprime set is now [9, 15, 25, 21, 35, 49].
7. **Increment to x = 9.**
8. Is 9 in nonprime? Yes.
   - The number 9 is not added to prime.
   - Generate composite numbers (9 * 3, 9 * 5, 9 * 7, 9 * 9).
   - Add 27, 45, 63, and 81 to nonprime.
   - The nonprime set is now [9, 15, 25, 21, 27, 35, 45, 49, 63, 81].

This process continues. The algorithm progressively *extends* the set of non-prime numbers in advance of its own progression. The fundamental concept is that any number *x* not contained in the nonprime set *is necessarily prime*.

3.0 Proof of Concept and Optimization

This section presents the development of the algorithm, from a basic proof-of-concept to a highly optimized function. The code for each level is included to demonstrate this evolution.

- **Level 1** establishes the core mathematical concept.
- **Level 2** introduces the first major logical optimization: skipping all even numbers after handling the prime '2'. This immediately doubles the algorithm's speed.
- **Level 3** refines this logic by introducing a statement, preventing the algorithm from calculating composite numbers larger than the maximum limit.
- **Level 4** marks a significant leap in performance by replacing the **Array** with a JavaScript **Set** for near-instant lookups. It also implements seed files allowing a user to halt and resume calculations, enabling the generation of primes into the hundreds of millions or billions over time.

3.1 Level 1: Proof of Concept

```
/**
 * Level 1 is a proof of concept and calculates prime numbers up to 1000.
 */
function level1() {
    const prime = [];
    const nonprime = [];

    for (let x = 2; x <= 1000; x++) {

        // Test x against the nonprime array.
             if (!nonprime.includes(x)) {
            // If not, add it to the prime array
            prime.push(x);
                    nonprime.push(x*x);
        }

        // Multiply x by every value in the prime array
        for (const p of prime) {
            const product = x * p;
            if (!nonprime.includes(product)) {
                nonprime.push(product);
            }
        }
    }

    return prime;
}
```

3.2 Level 2: Skip Even Numbers

```
/**
 * Level 2 skips even numbers and calculates prime numbers up to 1,000.
 * This version is optimized to only generate ODD composite numbers.
 */
function level2() {
    const prime = [2];
    // We only need to store odd non-primes,
       // as the loop will never check an even number.
    const nonprime = [];

       // Skip even numbers, starting with 3
    for (let x = 3; x <= 1000; x+=2) {

        // Test x against the nonprime array.
              if (!nonprime.includes(x)) {
            // If not, add it to the prime array
            prime.push(x);
            // x*x will always be odd, so we add it.
                     nonprime.push(x*x);
        }

        // Multiply x by every ODD value in the prime array
        // We start the loop at i=1 to skip prime[0], which is 2.
        for (let i = 1; i < prime.length; i++) {
            const p = prime[i]; // p will be 3, 5, 7, ...
            const product = x * p;

            // Only add the product if it's not already in the list
            // This is necessary because this loop runs for composite 'x' values,
            // creating duplicate products (e.g., 9*5 = 45 and 15*3 = 45).
            if (product <= 1000 && !nonprime.includes(product)) {
                nonprime.push(product);
            }
        }
    }

    return prime;
}
```

3.3 Level 3: Improved Efficiency

```
/**
 * Level 3 improved efficiency calculates prime numbers up to 10,000.
 * This version is optimized to only generate ODD composite numbers.
 */
function level3() {
    const prime = [2];
    // We only need to store odd non-primes,
       // as the loop will never check an even number.
    const nonprime = [];

    // Set maxNumber
    const maxNumber = 10000;

    // Loop through odd numbers up to maxNumber
    for (let x = 3; x <= maxNumber; x+=2) {

        // Test x against the nonprime array.
              if (!nonprime.includes(x)) {
            // If not, add it to the prime array
            prime.push(x);

            // Only add x*x if it's within the maxNumber limit
            const square = x * x;
            if (square <= maxNumber) {
                nonprime.push(square);
            }
        }

        // Optimization: Only generate new non-primes within the limit
        // Multiply x by every ODD value in the prime array
        // We start the loop at i=1 to skip prime[0], which is 2.
        for (let i = 1; i < prime.length; i++) {
            const p = prime[i]; // p will be 3, 5, 7, ...
                     const product = x * p;

                     // Only add products within the limit
                     if (product <= maxNumber && !nonprime.includes(product)) {
                            nonprime.push(product);
                     }
                     // If product is already too big, no need to check other primes
                     if (product > maxNumber) {
                            break;
                     }
              }
    }

    return prime;
}
```

3.4 Level 4: High-Performance and Seed File Implementation

The transition to Level 4 introduces two key improvements that enable high-performance, large-scale calculations. The first is a change in the data structure used for nonprime, and the second is the logic to make the calculation scalable.

The first improvement and most significant performance gain comes from replacing the nonprime Array with a JavaScript Set. In Levels 1-3, the nonprime.includes(x) check requires the computer to iterate through the array until it finds a match. As the nonprime array grows to millions of entries, this O(n) lookup becomes prohibitively slow. A Set, by contrast, uses a hash-based lookup, which is O(1). This means checking nonprime.has(x) takes the same near-instant amount of time whether the set contains one hundred entries or one hundred million. This single change is the most important software optimization in the algorithm.

The second improvement is the implementation of a seed file, which makes the algorithm scalable. The Level 4 function is designed to accept an existing array of primes (initialPrimes) as an argument. When provided, it performs a "catch-up" step, pre-populating the nonprime set with all the required multiples of those seed primes up to the new maxNumber. This allows a calculation to be saved and resumed. For example, a user can calculate all primes up to 10,000,000, save the result, and then load that file later to continue the calculation up to 20,000,000 without having to start from the beginning. This transforms the function from a simple script into a scalable tool.

```
/**
 * Level 4: Calculates prime numbers using an optional seed file and provides
progress updates.
 *
 * @param {number} maxNumber The new upper limit for prime number calculation.
 * @param {number[]} initialPrimes An array of primes from a seed file. Can be
empty.
 * @param {function} progressCallback A function to call with the current number
being processed.
 * @returns {number[]} The complete array of prime numbers up to maxNumber.
 */
function level4(maxNumber, initialPrimes, progressCallback) {
    const nonprime = new Set();
    const prime = [...initialPrimes];

    let startX = 3;
    if (prime.length > 0) {
        const lastPrime = prime[prime.length - 1];
        startX = lastPrime % 2 === 0 ? lastPrime + 1 : lastPrime + 2;
    } else {
        prime.push(2);
    }

    // **Pre-populate the nonprime set based on the seed primes.**
    // This is a crucial "catch-up" step.
    if (initialPrimes.length > 0) {
        progressCallback('Pre-calculating non-primes...');
        // We can skip prime[0] which is 2,
             //since our main loop only checks odd numbers.
        for (let i = 1; i < initialPrimes.length; i++) {
            const p = initialPrimes[i];

            // Find the first odd multiple of p that is >= startX
            let startMultiple = Math.floor(startX / p) * p;
            if (startMultiple < startX) {
                startMultiple += p;
            }
            if (startMultiple % 2 === 0) {
                startMultiple += p;
            }

            // Add all odd multiples of p up to the maxNumber
            for (let j = startMultiple; j <= maxNumber; j += (p * 2)) {
                nonprime.add(j);
            }
        }
    }

    // Main calculation loop
    for (let x = startX; x <= maxNumber; x += 2) {

        if (x % 1001 === 1) {
            progressCallback(x);
        }

        if (!nonprime.has(x)) {
            prime.push(x);
            // Optimization: Only need to mark multiples starting from x*x,
            // as smaller multiples would have been handled by smaller primes.
            for (let i = x * x; i <= maxNumber; i += (x * 2)) {
                nonprime.add(i);
            }
        }
    }

    return prime;
}
```

3.5 Link to the Public Repository

The complete source code for this project is available in a public GitHub repository. This repository contains the standalone JavaScript files for each of the four optimization levels (level1.js through level4.js). It also includes a runnable web application (primes.html, main.js, and styles.css) that provides a user interface for executing each level algorithm and managing and downloading seed files associated with Level 4. A README.md file provides instructions for running the application locally on a desktop computer.

The repository can be accessed at: https://github.com/wescoup/prime-number-generator

5.0 Conclusion

This paper presents a constructive method for generating prime numbers, offering an alternative to traditional trial division and sieving. By focusing on the direct generation of the composite set, it provides an understanding of the fundamental structure of prime and non-prime numbers.

This paper began by establishing the core algorithm, a constructive approach to generate the set of all composite numbers up to a predefined limit and defining its complement as the set of prime numbers. It then provided a simple proof of concept (Level 1), iteratively improved its efficiency through logical assumptions (Levels 2 & 3), and finally, applied software engineering principles to create a scalable and high-performance tool (Level 4). The result is a progression from a mathematical idea to a practical, working application.

The algorithm's implementation in JavaScript is intended to be accessible to a wide audience of programmers and developers, not just number theorists. The logic is presented in a common, high-level language, allowing for ready adaptation to other programming environments.