Constructing Carmichael Numbers

Mahesh Ramani

9/23/25

Introduction

In a general sense, Carmichael numbers are composite numbers that satisfy Fermat's Little Theorem for all integer bases. Fermat's Little Theorem states that if p is a prime number and a isn't divisible by p, then:

$$a^{p-1} \equiv 1 \pmod{p}$$

This is a pattern that all prime numbers follow. So, a common way to check if a number might be prime is to plug in random values of a and see if this congruence holds up.

However, just because all primes follow this rule doesn't mean only primes do. Some composite numbers also satisfy the congruence for certain values of a. If it holds for a few as, we call the number "probably prime" (a brilliantly straightforward name, I know).

But it gets worse: some composite numbers satisfy this congruence for every single value of a. These are Carmichael Numbers, and they're the reason a bare-bones Fermat primality test can't be fully trusted (and why nobody uses it for primality testing). This problem was taken so seriously that a mathematician named Richard Pinch went and calculated all Carmichael numbers up to 10^{21}, which is insane. Inspired by the challenge, I decided to try my hand at creating a little algorithm to find them myself (as in, generating all of these numbers up to a specified limit, n). This paper will walk through the mathematical backbone of how I did it.

1 Motivation and Korselt's Theorem

Carmichael numbers are really rare. Because they're so rare, simply iterating through every number from 1 to some huge power of ten would be a waste of my time and my computer's patience. The solution: I just make them up. "Constructing" would be a more appropriate term, however.

To construct them, I needed a common thread that all Carmichael numbers share. Luckily, mathematicians have already figured out some handy properties that we can use to our advantage: Korselt's Theorem.

You can read about the proof of the theorem on proofwiki.org (you should, it's a cool one), but for brevity's sake, I will not go into it. Here's what you need to know; Korselt's Theorem shows that a number n is Carmichael iff

- n is odd,
- n is composite,
- n is squarefree,

and for a given prime factor p of n:

$$p^2 \nmid n \qquad \text{and} \qquad p-1 \mid n-1.$$

2 Overview of the Constructive Approach

Now that we know that, let's glaze over the mathematical essence of my formula:

A Carmichael number n, like any integer, can be expressed by its unique prime factorization. Let's say our Carmichael number has k distinct prime factors. The goal of this algorithm is to generate these prime factors so that their product results in a Carmichael number.

We'll generate the primes in increasing order because it's a neat and logical way to avoid duplicates (remember that Korselt said p^2 cannot divide n for Carmichaels). So, for our prime factors:

$$p_1 < p_2 < \cdots < p_k$$

and their product must satisfy:

$$p_1 p_2 \cdots p_k \leq n.$$

Now, let's say we're in the middle of building this list and we're choosing the prime at position i, which we'll call p_i. We already have all the smaller primes (p_1 to p_{i-1}). Let's define the product of these known, smaller primes as:

$$P_{i-1} = p_1 p_2 \cdots p_{i-1}.$$

The smallest possible value for the product of the remaining primes (p_i up to p_k) would be if they were the next $k-(i-1)$ smallest possible primes. However, calculating that on the fly for every step seemed like it could get inefficient (someone could check me on that, though).

So, let's be a little lazy but effective: we know that the product of p_i and all the larger primes after it must be at least p_i raised to the power of how many primes are left. Let $r = k - i + 1$ be the number of primes we still need to choose (including p_i itself).

This gives us a constraint:

$$P_{i-1}\,(p_i)^r \leq n.$$

Solving for p_i, we get its upper bound:

$$p_i \leq \left(\frac{n}{P_{i-1}}\right)^{1/r}.$$

As for the lower bound, it's simple: p_i just has to be greater than the previous prime, p_{i-1}.

Now, we skipped a step: how do we even choose the first prime, p_1? Luckily, our formula works perfectly for this. If we plug in $i = 1$, then the product of the previous primes, P_0, is just 1 (since there are no primes before it). This gives us a nice, clean upper bound:

$$p_1 \leq n^{1/k}.$$

The lower bound is straightforward, too. Since Carmichael numbers are odd and composite, we can't use 2. So, p_1 must be at least 3.

3 Bounds on k and the Search Strategy

The final piece of the pruning puzzle is to figure out the possible range for k itself. The lower bound is simple: by definition, a Carmichael number must have at least three distinct prime factors, so $k \geq 3$.

For the upper bound, we can be lazy again. We can find the maximum number of prime factors k could have by multiplying the smallest odd primes $(3, 5, 7, 11, \ldots)$ until the product exceeds our limit n. The count of primes we used gives us a safe upper bound for k.

So, the overall process for our algorithm is to loop over all possible values of k from the lower bound to the upper bound. For each k, we then use our recursive prime-generating formula to build valid tuples of primes that could form a Carmichael number up to n.

4 The Last Prime and Modular Conditions

Now, let's focus on the last prime in our tuple, which we'll call p_k. Let P_{k-1} be the product of all the primes before it (p_1 to p_{k-1}).

To ensure our number $n = P_{k-1}p_k$ is a Carmichael number, it must satisfy Korselt's criterion. One part of this criterion requires that for each prime factor p_i:

$$n \equiv 1 \pmod{p_i - 1}.$$

Let's apply this for the first prime factor, p_1:

$$n \equiv 1 \pmod{p_1 - 1}.$$

Substituting $n = P_{k-1}p_k$:

$$P_{k-1}p_k \equiv 1 \pmod{p_1 - 1}.$$

We can factor P_{k-1} as p_1P', where P' is the product of the remaining primes (p_2 to p_{k-1}). This gives us:

$$p_1P'p_k \equiv 1 \pmod{p_1 - 1}.$$

However, since $p_1 \equiv 1 \pmod{p_1 - 1}$, this simplifies to:

$$P'p_k \equiv 1 \pmod{p_1 - 1}.$$

Therefore, p_k must be the modular inverse of P' modulo $p_1 - 1$:

$$p_k \equiv a \pmod{p_1 - 1}$$

where a is the modular inverse of P' modulo $p_1 - 1$ (i.e., $a \cdot P' \equiv 1 \pmod{p_1 - 1}$). Sometimes, a doesn't exist, so we discard the tuple if we find that in the code.

But here's the catch: we must satisfy this congruence not just for p_1, but for every prime factor in P_{k-1}. This means we end up with a system of modular congruences that p_k must simultaneously satisfy.

So, how do we handle a whole system of these congruences? We solve for p_k using the generalized Chinese Remainder Theorem (CRT). This bundles all those individual modulo conditions into one neat package:

$$p_k \equiv a \pmod{M}$$

where M is the least common multiple of all the $(p_i - 1)$ terms for the primes in our set.

Note: the code's `crt_pair` handles non-coprime moduli by checking consistency (it uses extended gcd). Unlike the usual CRT (where moduli must be coprime), here we allow overlaps, so we need to check consistency with gcds.

This means the possible solutions for p_k are all numbers of the form $a + tM$, where t is an integer. We start generating candidates by plugging in $t = 0, 1, 2, \ldots$, making sure each candidate is greater than our largest prime so far.

For each candidate r that is also prime, we have to do one final check. Remember, for n to be a Carmichael number, Korselt's criterion must hold for r itself! This means we need to check:

$$P \equiv 1 \pmod{r-1}$$

where P is the product of all the primes in our set. In other words, $r - 1$ must divide $P - 1$. If this holds true, and the total product $P \cdot r$ is within our limit n, then congratulations—we've built a Carmichael number!

And of course, we have to make sure r isn't already in our set of primes, but the way we generate it takes care of that.

The cool part is that this whole process isn't just for the final prime. We can use the same trick at any step: pick $k - 1$ primes, use CRT to find candidates for the k-th prime, and check that last divisibility condition. It's a powerful tool.

5 Implementation Notes

*E*very case where we need to check if a number is prime comes from `is_prime()`, which is a deterministic Miller–Rabin primality tester for 64-bit integers. Pushing N large enough to where it may exceed the 64-bit safe range, the test must be adjusted. But it should work for $N \leq 10^{18}$.

Also, I glazed over a lot of the nitty-gritty of how the formula works, as I just wanted to write about the math that interested me behind it. Please feel free to check out the code itself!

6 Complexity and Final Thoughts

I thought the code was worth sharing since I ended up with an optimistic time complexity of $\Theta\left(\frac{N^{2/3}}{\log N}\right) \cdot C$ for practical cases and space complexity $O(\sqrt{N})$ for prime storage. It's generally efficient, and it could be improved even more if you translate it into C.