Constructing Carmichael Numbers

Mahesh Ramani
mahesh.ramani.iyer@gmail.com

September 25, 2025

Abstract

This note presents a summary of a constructive algorithm for generating Carmichael numbers up to a bound N. I motivate the construction with Korselt's theorem, summarize the modular constraints that prime factors must satisfy, and show how to use a generalized Chinese Remainder Theorem (CRT) to recover the final prime factor. I describe the implementation details (deterministic Miller–Rabin for 64-bit integers, a simple sieve, and auxiliary number-theory routines), analyze time and space trade-offs in Big-O notation, and compare the practical behavior of the implementation against naive enumerations. The approach exploits heavy pruning via arithmetic bounds and CRT congruences to make the search feasible up to moderately large N.

1 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:

$$a^{\,p-1} \equiv 1 \pmod{p} \qquad \text{for every } \gcd(p, n) = 1.$$

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).

Korselt's theorem gives a constructive handle which helps us begin:

> An integer $n > 1$ is a Carmichael number if and only if n is odd, squarefree, composite, and for every prime $p \mid n$ we have $p - 1 \mid n - 1$.

That last divisibility condition - $p-1 \mid n-1$ - is exactly what we exploit. If $n = p_1 p_2 \cdots p_k$ with distinct primes p_i, then each condition $p_i - 1 \mid n-1$ imposes a modular constraint on any missing factor(s). The algorithm below builds prime factor tuples in increasing order, uses multiplicative bounds to prune impossible branches, and merges the resulting congruences (from Korselt's theorem) with generalized CRT to recover the final prime factor.

2 Notation and preliminaries

- N (sometimes `upper_bound`) is the search limit: we look for Carmichael numbers $\leq N$.
- k is the number of distinct prime factors of a candidate (Korselt implies $k \geq 3$). r is the greatest prime factor of a candidate.
- $p_1 < p_2 < \cdots < p_k$ are the distinct prime factors; for $i \geq 1$ write $P_i = \prod_{j=1}^{i} p_j$ and $P_0 = 1$.
- `is_prime(n)` denotes the deterministic Miller–Rabin used in the implementation; `crt`/`crt_pair` are the incremental CRT helpers that check gcd-consistency.

3 Constructive Approach

Primes are chosen in strict increasing order:

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

This is convenient and cheap:

1. Squarefreeness is automatic (no repeated primes).
2. It yields monotone multiplicative bounds: with P_{i-1} known, we get tight caps for p_i, so the search explores only a narrow interval instead of a huge range.

If we've fixed $p_1, \ldots, p_{i-1}$ with product P_{i-1}, there remain $r = k - i + 1$ primes (including p_i). Since the remaining product more than p_i^r, we have

$$P_{i-1} \cdot p_i^r \leq N,$$

hence

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

with the trivial lower bound $p_i > p_{i-1}$. These bounds drastically shrink the recursion tree. For $i = 1$ this reproduces $p_1 \leq N^{1/k}$. A cheap upper bound for k follows by multiplying the smallest odd primes until the product exceeds N.

And to restate:

1. by the time we pick the final factor $r = p_k$, all smaller primes have been considered (used or skipped), so duplicates cannot occur;
2. we enforce $r > p_{k-1}$ when accepting a candidate, guaranteeing strict ordering.

4 Modular constraints and the generalized CRT

Korselt's condition $p - 1 \mid n - 1$ for each prime $p \mid n$ becomes a system of congruences when $n = P_{k-1} \cdot r$. For each $p_i \mid P_{k-1}$ set $m_i = p_i - 1$ and $Q_i = P_{k-1}/p_i$; then

$$n \equiv 1 \pmod{m_i} \quad \Longrightarrow \quad Q_i \cdot p_i \cdot r \equiv 1 \pmod{m_i}.$$

Since $p_i \equiv 1 \pmod{m_i}$ this simplifies to

$$Q_i \cdot r \equiv 1 \pmod{m_i},$$

so whenever $\gcd(Q_i, m_i) = 1$ we get $r \equiv a_i \pmod{m_i}$ where a_i is the inverse of Q_i modulo m_i.

Collecting these congruences for $i = 1, \ldots, k-1$ produces

$$r \equiv a_1 \pmod{m_1}, \quad r \equiv a_2 \pmod{m_2}, \ \ldots, \quad r \equiv a_{k-1} \pmod{m_{k-1}}.$$

An incremental generalized CRT merges these, checking gcd-consistency:

$$(a_j - a_i) \equiv 0 \pmod{\gcd(m_i, m_j)}$$

for each pair. If merging succeeds we obtain a combined congruence $r \equiv a \pmod{M}$ (with M essentially the lcm of the m_i). Candidate final primes are then

$$r = a + tM, \qquad t \in \mathbb{Z}_{\geq 0},$$

subject to $r > p_{k-1}$ and $P_{k-1} \cdot r \leq N$. We enumerate such r, test primality, and additionally verify $r - 1 \mid P_{k-1} - 1$ (the final Korselt check). If all checks pass, $P_{k-1} \cdot r$ is recorded as a Carmichael number.

5 Algorithm (high-level pseudocode)

Below is the high-level procedure

```
for k from 3 to k_max:
  compute safe bounds for each p_i:
    p_{i-1} < p_i <= (N / P_{i-1})^(1 / (k-i+1))
  recursively pick p1 < p2 < ... < p_{k-1} from primes:
    compute congruences r  a_i (mod m_i) with m_i = p_i - 1
    merge congruences via generalized CRT -> r  a (mod M)
    enumerate r = a + t*M with r > p_{k-1} and P_{k-1}*r <= N:
      if r is prime and (P_{k-1}-1) % (r-1) == 0:
        record P_{k-1}*r as a Carmichael number
```

The implementation also computes minimal possible products for remaining primes to prune early, and rejects branches when modular inverses do not exist. For k=3, we're able to simplify the CRT work needed and the upper/lower bounds for each prime factor are trivialized. Therefore, we treat that case separately in the code.

6 Complexity analysis

6.1 Sieve and precomputation

A standard sieve builds the prime list up to a chosen *sieve upper*. Memory cost depends on that parameter (an engineering choice; in the code it's chosen relative to $N^{1/3}$ but adjustable).

6.2 Enumeration cost and pruning

Naive enumeration for $k = 3$ gives roughly $O(N^{5/6}/\log^2 N)$ candidate pairs, but multiplicative root bounds, minimal-product pruning, modular-inverse checks, and CRT consistency reject the majority of these before primality testing. Empirically the runtime behaves considerably better than the naive bound for moderate N; a heuristic practical runtime near $\tilde{O}(N^{2/3}) \cdot C$ was observed where C collects CRT and MR constants (this is heuristic and implementation-dependent).

6.3 Per-candidate cost

Primality testing uses deterministic Miller–Rabin (constant witness set in the targeted range), costing $O(\log^3 N)$ per test with standard modular exponentiation. CRT merges cost GCDs and small multiplications and are cheap relative to repeated primality tests; many branches die before full merging.

7 Implementation notes

- **Language & libraries:** Python 3. The implementation contains a sieve (`sieve_primes`), extended gcd / modular inverse (`egcd`, `modinv`), generalized CRT (`crt_pair`, `crt`), and deterministic Miller–Rabin (`is_prime`). Special handling for $k = 3$ speeds up the most common case.
- **Deterministic primality:** the Miller–Rabin witness set is chosen to be deterministic for 64-bit integers; for larger ranges use more bases or a provable method.
- **Adjustable sieve bound:** the sieve upper is tunable and can be increased to search larger N.
- **Final verification:** after collecting candidates the code factors and verifies each result (squarefree, at least three prime factors, and all Korselt conditions) to avoid false positives.

8 Trade-offs and Practice

- Translating divisibility constraints into a single arithmetic progression via CRT simplifies the search for the final prime.
- Per-prime root bounds are cheap and effective: they bluntly eliminate impossible branches and keep the recursion manageable.
- Allowing non-coprime moduli increases potential consistency cases but requires rigorous gcd-consistency checks. The generalized CRT does this reliably.
- Constants (sieve size, CRT costs, number of t-values) dominate runtime in practice.

9 Conclusion

The algorithm builds prime-factor tuples in increasing order, uses the bound

$$p_i \leq \left(\frac{N}{P_{i-1}} \right)^{1/(k-i+1)}$$

to prune heavily, and merges modular constraints into $r \equiv a \pmod{M}$ using a generalized CRT. Enumerating that progression and applying deterministic primality checks yields Carmichael numbers up to N. This combination of multiplicative pruning and CRT-based sparsification makes a constructive search practical for reasonably large N while avoiding duplicates by construction.