An Algorithm for Computing Highly Composite Numbers

Kiran S. Kedlaya

September 12, 2016

1 Introduction

A *highly composite number* is a positive integer with more divisors than any smaller positive integer. In other words, if $\tau(n)$ denotes the number of divisors of n, then n is highly composite if $\tau(m) < \tau(n)$ for all $m < n$. The highly composite numbers (HCNs) were introduced by Ramanujan [3], who used them to study the asymptotic growth of the τ-function. Subsequent investigators of these numbers include Erdős [1] and Nicolas [2].

In order to guess asymptotic properties of the highly composite numbers, it helps to be able to compute them efficiently. Robin [4] gave an algorithm for computing HCNs based on the notion of "bénéfice" (benefit). The purpose of this note is to describe another such algorithm, which has the advantages of being fairly simple as well as reasonably efficient.

2 The Algorithm

The key to the method is the notion of a *highly composite k-product* (abbreviated HCP_k), defined to be a number with k distinct prime factors having more divisors than any smaller number with k distinct prime factors. The following observations are immediate consequences of the definition:

- Every HCP_k is of the form $p_1^{a_1} \dots p_k^{a_k}$, where p_i is the i-th prime and $a_1 \geq a_2 \geq \dots \geq a_k > 0$.
- If $p_1^{a_1} \dots p_k^{a_k}$ is an HCP_k, then $p_1^{a_1} \dots p_{k-1}^{a_{k-1}}$ is an HCP_{k-1}.
- Every HCN with exactly k prime factors is an HCP_k.

Thus given a sufficiently long list of HCP_{k-1}'s, one can construct a list of HCP_k's as follows. Given an HCP_k n, for successive values of j, find the smallest HCP_{k-1} m such that $(j+1)\tau(m) > \tau(n)$. The next HCP_k is then the minimum of mp_k^j over all j. (Clearly once we encounter a value of j for which $m = f(k-1, 1)$, we need not consider larger j.)

This translates into a simple algorithm as follows. Let $f(k, n)$ denote the n-th HCP_k and $d(k, n) = \tau(f(k, n))$. The above discussion reduces the computation of f to the computation of functions $g(k, n)$ and $h(k, n)$ for $n \geq 1$ and $k \geq 2$ such that

$$f(k, n) = p_k^{g(k,n)} f(k-1, h(k, n)), \quad d(k, n) = (g(k, n) + 1) d(k-1, h(k, n)).$$

We can ignore $k = 1$ since clearly $f(1, n) = 2^n$.

Algorithm 1: Computing HCP$_k$'s
Step 1: If $n = 1$, let $g(k, n) = h(k, n) = 1$ and STOP. Otherwise, let $r = 2f(k, n-1)$ and $j = 1$.
Step 2: Find the smallest integer s for which either $(j + 1)d(k - 1, s) > d(k, n - 1)$ or $p_k^j f(k - 1, s) > r$. If the latter fails to hold, let $r = p_k^j f(k - 1, s)$, $e_k = j$, and $m = s$.
Step 3: If $s > 1$, add 1 to j and return to Step 2. Otherwise, let $g(k, n) = e_k$, $h(k, n) = m$ and STOP.

The HCNs can be found in a table of HCPs by a process parallel to Algorithm 1. Namely, let $H(n)$ denote the n-th HCN. To find $H(n)$, for each k, find the smallest HCP$_k$ m with more divisors than $H(n - 1)$; the smallest of these is $H(n)$. (As in Algorithm 1, once a value of k is found such that $m = f(k, 1)$, we need not consider larger k.)

Algorithm 2: Computing HCN's
Step 1: If $n = 1$, let $H(n) = 1$ and STOP. Otherwise, let $r = 2H(n - 1)$ and $k = 1$.
Step 2: Find the smallest integer s for which either $d(k, s) > \tau(H(n - 1))$ or $f(k, s) \geq r$. If the latter fails to hold, let $r = f(k, s)$.
Step 3: If $s > 1$, add 1 to k and return to Step 2. Otherwise, let $H(n) = r$ and STOP.

3 Implementation

While the algorithms are simple enough to describe, making them run efficiently is a bit trickier. In this section, we describe some modifications we have made to improve performance.

For successive values of k, we use Algorithm 1 to generate a list of the values of $d(k, n)$, $f(k, n)$, $g(k, n)$, $h(k, n)$; we then use Algorithm 2 to locate HCNs in these lists. The maximum length of a list, the number of lists, and the number of HCNs are specified at runtime, though a list is truncated before the maximum length if an uncomputed value from a previous list is needed.

In Step 2 of either algorithm, we are asked to find the smallest s with a given property; we can profit from the fact that with each pass through the algorithm, this s is getting larger. To be precise, in Algorithm 1, for fixed k and j, the value of s is never decreasing, while in Algorithm 2, for fixed k the value of s is never decreasing. Hence by keeping track of the last values used and searching from that point instead of from 1, we save a great deal of time.

A second modification, which is easy to implement but slightly complicated conceptually, involves decreasing the search space at Step 1. We describe this first for Algorithm 2, where the necessary modification is fairly simple.

Proposition 1 *If n is an HCN with k distinct prime factors, then $n \leq p_{k+1}^{2k}$.*

PROOF: Factor n as $p_i^{e_1} \dots p_k^{e_k}$ and suppose $n > p_{k+1}^{2k}$. Then for some i, $p_i^{e_i} > p_{k+1}^2$. Let $m = \lceil \log p_{k+1} / \log p_i \rceil$; then $e_i > 2 \log_{p_i} p_{k+1} \geq 2m - 1$. But this means that np_{k+1}/p_i^m is an integer less than n with $\tau(n)2(e_i - m + 1)/(e_i + 1) \geq \tau(n)$ divisors, so n is not an HCN. □

Therefore in Step 1, we may set k to be the smallest integer such that $n \leq p_{k+1}^{2k}$ rather than 1, eliminating deep searches in lists that will not yield any more HCNs.

The corresponding modification to Algorithm 1 requires a lower bound on the exponent of p_k for a large HCP_k. Such a bound can be derived by modifying the above argument, but we get a much better estimate by a different approach.

Proposition 2 *For any $n, k \in \mathbb{N}$, there exists $t \leq n$ with at most k prime factors such that*

$$\tau(t) \geq \left(\frac{\log n}{k}\right)^k \prod_{i=1}^{k} \frac{1}{\log p_k}.$$

Proof: Let $\lambda_i = \log n/(k \log p_i)$ and put $e_i = \lfloor \lambda_i \rfloor$ and $t = \prod p_i^{e_i}$. Then

$$\tau(t) = \prod (e_i + 1) \geq \prod \lambda_i = \left(\frac{\log n}{k}\right)^k \prod_{i=1}^{k} \frac{1}{\log p_k}.$$

□

Proposition 3 *Suppose*

$$\frac{(\log n)^k}{(\log n + \log p_1 + \ldots + \log p_{k-1})^{k-1}} > ke(\ell + 1) \log p_k.$$

If $m = p_1^{e_1} \ldots p_k^{e_k}$ is an HCP_k greater than n, then $e_k \geq \ell$.

Proof: As the right side is increasing in ℓ, it suffices by induction to prove that $e_k \neq \ell$. The left side is increasing in $\log n$ (factor off $\log n$ and the rest is obviously increasing), so the assumed inequality still holds with m in place of n. If $e_k = \ell$, we have by the AM-GM inequality,

$$\left(\frac{\log m - \ell \log p_k + \log p_1 + \ldots + \log p_{k-1}}{k-1}\right)^{k-1} \geq \prod_{i=1}^{k-1} [e_i + 1] \log p_i = \frac{\tau(m)}{\ell + 1} \prod_{i=1}^{k-1} \log p_i.$$

On the other hand, by the previous lemma, there exists $t \leq m$ such that

$$\begin{aligned}
\tau(t) &\geq \left(\frac{\log m}{k}\right)^k \prod_{i=1}^{k} \frac{1}{\log p_k} \\
&> \frac{(\log m + \log p_1 + \ldots + \log p_{k-1})^{k-1}}{k^k} ke(\ell + 1) \log p_k \prod_{i=1}^{k} \frac{1}{\log p_k} \\
&\geq \left(\frac{\log m - \ell \log p_k + \log p_1 + \ldots + \log p_{k-1}}{k-1}\right)^{k-1} e(\ell + 1) \left(\frac{k-1}{k}\right)^{k-1} \prod_{i=1}^{k-1} \frac{1}{\log p_k} \\
&\geq \tau(m),
\end{aligned}$$

using the fact that $e > [k/(k-1)]^{k-1}$ for all k. Hence m cannot be an HCP_k. □

With these modifications, we have recreated Robin's table of 5000 highly composite numbers in several minutes on a Sun workstation. Ramanujan's table of 102 HCNs appears almost instantly (note that his table is missing the HCN 293318625600 between the 85th and 86th terms). These tables and the C code of the implementation described above can be obtained from the author's WWW site INSERT-URL.

References

[1] P. Erdős, On highly composite numbers, *J. London Math. Soc.* **19** (1944) 130-133.

[2] J.-L. Nicolas, Nombres hautement composés, *Acta Arith.* **49** (1988) 395-412.

[3] S. Ramanujan, Highly composite numbers, *Proc. Lond. Math. Soc.* (2) **14** (1915) 347-409.

[4] G. Robin, Méthods d'optimisation pour un problème de théorie des nombres, *R.A.I.R.O. Informatique théoretique* **17** (1983) 239-247.