Artem Bolshakov

The Natural Density of Abundant Numbers

The natural density of abundant numbers is a value to whose calculation primitive abundant numbers yield themselves most easily; all we need to find the natural density of all numbers divisible by at least one of them.

If only it was that easy; the most efficient algorithm that I was able to find for finding the natural density of the set M of numbers divisible by at least 1 element of a given set S takes exponential memory with respect to the size of S. At present, I cannot get a much better lower bound than 0.2402.

Even so, however, I did optimize my algorithm so much, it is difficult to see why it actually computes the natural density of M(S). Therefore, it is necessary to first go through a discussion of calculating that value.

The most obvious method is to take $1 - (\Pi_{x \varepsilon S}(x-1)/x)$. However, this only works if all elements of S are coprime with each other; if they are not, the fact of a number being divisible by one number increases its probability of being divisible by the other; thus, the set of numbers divisible by the first and divisible by the second will have a greater natural density.

So, the way we might begin is by computing $\Sigma_{x \varepsilon S}\ (1/x)$, because the natural density of the multiples of x is 1/x. However, we have counted the multiples of each of the pairwise LCMs twice (1 for each of the numbers whose LCMs they are); thus we need to subtract them out. Denoting $T_1 = S$ and T_n as the set of n-wise LCMs of S, we now have $1*(\Sigma_{x \varepsilon T1}\ (1/x)) + (-1)*(\Sigma_{x \varepsilon T2}\ (1/x))$. We now note that this calculation, however, ignores all multiples of elements of T_3; they are added in three times as multiples of elements of T_1 and subtracted out three times as multiples of elements of T_2; thus, we need to add them back in: $1*(\Sigma_{x \varepsilon T1}\ (1/x)) + (-1)*(\Sigma_{x \varepsilon T2}\ (1/x)) + 1*(\Sigma_{x \varepsilon T3}\ (1/x))$.

Continuing in this way, we find that we have to separately consider all multiples of elements of all sets T: the quantity of multiples of elements of T_n counted depends on the way we added in or subtracted $\Sigma_{x \varepsilon Tn-1}\ (1/x)$. Thus, if there are n elements in S, we can write the natural density of multiples

of any elements of S (hence denoted probdiv(S)) as probdiv(S) = $\Sigma_{k=1}^{n} a_k * (\Sigma_{x \in Tk} (1/x))$ for some set of coefficients a_k (independent of n, as it only looks at all of the multiples of elements of all the T's below T_k). The question now becomes, how can we find these a_k's?

Well, for every set of multiples of x, for x in any T, we want its natural density to be included exactly once. Thus, a_k = 1 - [the number of times the natural density of the set of multiples of x has already been included]. Now, for any T_i, i<k, we have that the set of multiples of $\binom{k}{i}$, or k choose i, elements of T_i already included the set of multiples of x for any x ϵ T_k. Thus, we can write a_k recursively as $a_k = 1 - \Sigma_{i=1}^{k-1} a_i * \binom{k}{i}$.

At first, I assumed that this recursive form was the very best expression for a_k. However, when I got a python program to compute the first several of these values, I was surprised to see a consecutive series of 1's and -1's. I now assert that $a_k = (-1)^{k+1}$.

We may prove this by mathematical induction. The base is trivial, and has already been given. Now, for the step, let us assume that, for all i<k, $a_i = (-1)^{i+1}$. There are two cases: k is odd, and k is even.

The first case is fairly simple:

$\Sigma_{i=1}^{k-1} a_i * \binom{k}{i} = \Sigma_{i=1}^{k-1} (-1)^{i+1} * \binom{k}{i} = \Sigma_{j=1}^{(k-1)/2} ((-1)^{j+1} * \binom{k}{j} + (-1)^{k-j+1} * \binom{k}{k-j}) =$

$\Sigma_{j=1}^{(k-1)/2} ((-1)^{j+1} * \binom{k}{j} + (-1)^{j} * \binom{k}{j}) = \Sigma_{j=1}^{(k-1)/2} 0 = 0$

Thus, $a_k = 1 - 0 = 1 = (-1)^{k+1}$ for odd k, QED.

The second is a little more complex. Here, however, we can use the convenient property of Pascal's triangle that $\binom{k}{i} = \binom{k-1}{i-1} + \binom{k-1}{i}$. Now, due to the sign of the entire sum, $\binom{k-1}{i}$ has one sign when part of the sum generating $\binom{k}{i}$, but the *other* sign when part of the sum generating $\binom{k}{i+1}$. Thus, with the exception of $\binom{k-1}{0}$ (part of the sum generating $\binom{k}{1}$, thus being positive), and $\binom{k-1}{k-1}$ (part of the sum generating $\binom{k}{k-1}$, also positive due to the fact that, for even k, k -1 is odd), all $\binom{k-1}{i}$ cancel each other out. Thus, we have $\Sigma_{i=1}^{k-1} a_i * \binom{k}{i} = \Sigma_{i=1}^{k-1} (-1)^{i+1} * \binom{k}{i} = \binom{k-1}{0} + \binom{k-1}{k-1} = 1 + 1 = 2$, and thus $a_k = 1 - 2 = -1 = (-1)^{k+1}$ for even k, QED.

Thus, we have proved that probdiv(S) = $\Sigma_{k=1}^{n} (-1)^{k+1} (\Sigma_{x \varepsilon Tk} (1/x))$. This is precisely what the probdiv function in the module calc.py calculates, as before affirmed. Notably, the fact that we do not need to calculate the exact k of the T generating an LCM, only whether it is even or odd, allows for a much more optimal function.

This is exactly what my function does. I have provided two versions: one in the module "calc2", is more accurate, but takes up not only exponential time but also exponential memory, as it stores all intermediate values. The one in the module "calc3" does not store intermediate values, and hence may be used on arbitrarily large lists of numbers; but it still takes up far too much time.

I have also provided a third version, in the module "calc4", which uses the fact that the pattern of "divisible" or "not divisible" repeats modulo LCM of the list, and just brute forces the calculation up to the LCM of the file. This module is more convenient when the numbers share many divisors; it takes much more time for lists that have many coprime elements. Calculations show that, for large numbers of primitive abundant numbers, this method would save time; however, for all quantities where it is practical to calculate at all, the other modules are (or would be) much, much faster.

These are, unfortunately, the best algorithms that I was able to find so far. If any mathematicians find a more efficient formula for the natural density of M(S), please let me know; I will then be able to code that, and actually use that formula to find original results. Nevertheless, this inefficient algorithm at least demonstrates the potential utility of primitive abundant numbers; given enough time, they have the potential for finding arbitrarily good lower bounds on the natural density of abundant numbers, whose exact value is, at present, an enigma for mathematicians.

Applications so far: the natural density of numbers divisible by any of the first seventeen (all < 1000) primitive abundant numbers is ~ 0.240; the algorithm takes several seconds. For the first 35 (all < 3000), the value is ~ 0.2427 and takes several days. Calculations show that the first 102 (all < 10000) would take circa a quadrillion years. Unfortunately, none of these results are groundbreaking yet.