INTEGER COMPLEXITY: ALGORITHMS AND COMPUTATIONAL RESULTS

Harry Altman
Department of Mathematics, University of Michigan, Ann Arbor, Michigan
harry.j.altman@gmail.com

Received: 3/24/17, Revised: 12/21/17, Accepted: 5/10/18, Published: 5/18/18

Abstract
Define $\|n\|$ to be the *complexity* of n, the smallest number of ones needed to write n using an arbitrary combination of addition and multiplication. Define n to be *stable* if for all $k \geq 0$, we have $\|3^k n\| = \|n\| + 3k$. In a recent paper, this author and Zelinsky showed that for any n, there exists some $K = K(n)$ such that $3^K n$ is stable; however, the proof there provided no upper bound on $K(n)$ or any way of computing it. In this paper, we describe an algorithm for computing $K(n)$, and thereby also show that the set of stable numbers is a computable set. The algorithm is based on considering the *defect* of a number, defined by $\delta(n) := \|n\| - 3\log_3 n$, building on the methods presented in a recent article of this author. As a side benefit, this algorithm also happens to allow fast evaluation of the complexities of powers of 2; we use it to verify that $\|2^k 3^\ell\| = 2k + 3\ell$ for $k \leq 48$ and arbitrary ℓ (excluding the case $k = \ell = 0$), providing more evidence for the conjecture that $\|2^k 3^\ell\| = 2k + 3\ell$ whenever k and ℓ are not both zero. An implementation of these algorithms in Haskell is available.

1. Introduction

The *complexity* of a natural number n is the least number of 1's needed to write it using any combination of addition and multiplication, with the order of the operations specified using parentheses grouped in any legal nesting. For instance, $n = 11$ has a complexity of 8, since it can be written using 8 ones as

$$11 = (1 + 1 + 1)(1 + 1 + 1) + 1 + 1,$$

but not with any fewer than 8. This notion was implicitly introduced in 1953 by Kurt Mahler and Jan Popken [21]; they actually considered an inverse function, the size of the largest number representable using k copies of the number 1. (More generally, they considered the same question for representations using k copies of a positive real number x.) Integer complexity was explicitly studied by John Selfridge, and was later popularized by Richard Guy [16, 17]. Following J. Arias de Reyna [8] we will denote the complexity of n by $\|n\|$.

Integer complexity is approximately logarithmic; it satisfies the bounds

$$3\log_3 n = \frac{3}{\log 3}\log n \le \|n\| \le \frac{3}{\log 2}\log n, \qquad n > 1.$$

The lower bound can be deduced from the result of Mahler and Popken, and was explicitly proved by John Selfridge [16]. It is attained with equality for $n = 3^k$ for all $k \ge 1$. The upper bound can be obtained by writing n in binary and finding a representation using Horner's algorithm. It is not sharp, and the constant $\frac{3}{\log 2}$ can be improved for large n [26].

One can compute $\|n\|$ via dynamic programming, since $\|1\| = 1$, and for $n > 1$, one has

$$\|n\| = \min_{\substack{a,b<n\in\mathbb{N} \\ a+b=n \text{ or } ab=n}} (\|a\| + \|b\|).$$

This yields an algorithm for computing $\|n\|$ that runs in time $\Theta(n^2)$; in the multiplication case, one needs to check $a \le \sqrt{n}$, and, naïvely, in the addition case, one needs to check $a \le n/2$. However, Srinivas and Shankar [23] showed that the upper bound on the addition case can be improved, lowering the time required to $O(n^{\log_2 3})$, by taking advantage of the inequality $\|n\| \ge 3\log_3 n$ to rule out cases when a is too large. Arias de Reyna and Van de Lune [9] took this further and showed that it could be computed in time $O(n^\alpha)$, where

$$\alpha = \frac{\log(3^6 2^{-10}(30557189 + 21079056\sqrt[3]{3} + 14571397\sqrt[3]{9}))}{\log(2^{10}3^7)} < 1.231;$$

this remains the best known algorithm for computing $\|n\|$ for general n.

The notion of integer complexity is similar in spirit but different in detail from the better known measure of *addition chain length*, which has application to computation of powers, and which is discussed in detail in Knuth [20, Sect. 4.6.3]. See also [1] for some interesting analogies between them; we will discuss this further in Section 2.2.

1.1. Stability Considerations

One of the easiest cases of complexity to determine is powers of 3; for any $k \ge 1$, one has $\|3^k\| = 3k$. It is clear that $\|3^k\| \le 3k$ for any $k \ge 1$, and the reverse inequality follows from Equation (1).

The fact that $\|3^k\| = 3k$ holds for all $k \ge 1$ might prompt one to ask whether in general it is true that $\|3n\| = \|n\| + 3$. This is false for $n = 1$, but it does not seem an unreasonable guess for $n > 1$. Nonetheless, this does not hold; the next smallest counterexample is $n = 107$, where $\|107\| = 16$ but $\|321\| = 18$. Indeed, not only do there exist n for which $\|3n\| < \|3n\| + 3$, there are n for which $\|3n\| < \|n\|$; one example is $n = 4721323$. Still, this guess can be rescued. Let us first make a definition.

Definition 1. A number m is called *stable* if $\|3^k m\| = 3k + \|m\|$ holds for every $k \geq 0$. Otherwise it is called *unstable*.

In [7], this author and Zelinsky showed the following theorem.

Theorem 1. *For any natural number n, there exists $K \geq 0$ such that $3^K n$ is stable. That is to say, there exists a minimal $K := K(n)$ such that for any $k \geq K$,*

$$\|3^k n\| = 3(k - K) + \|3^K n\|.$$

This can be seen as a "rescue" of the incorrect guess that $\|3n\| = \|n\| + 3$ always. With this theorem, we can make the following definition.

Definition 2. Given $n \in \mathbb{N}$, define $K(n)$, the *stabilization length* of n, to be the smallest k such that $3^k n$ is stable.

We can also define the notion of the *stable complexity* of n (see [2]), which is, intuitively, what the complexity of n would be "if n were stable".

Definition 3. For a positive integer n, we define the *stable complexity* of n, denoted $\|n\|_{st}$, to be $\|3^k n\| - 3k$ for any k such that $3^k n$ is stable. This is well-defined; if $3^k n$ and $3^\ell n$ are both stable, say with $k \leq \ell$, then

$$\|3^k n\| - 3k = 3(k - \ell) + \|3^\ell n\| - 3k = \|3^\ell n\| - 3\ell.$$

The paper [7], while proving the existence of $K(n)$, gave no upper bound on $K(n)$ or indeed any way of computing it. Certainly one cannot compute whether or not n is stable simply by computing for all k the complexity of $3^k n$; one can guarantee that n is unstable by such computations, but never that it is stable. And it is not clear that $\|n\|_{st}$, though it has been a useful object of study in [2], can actually be computed.

1.2. Main Result

We state the main result.

Theorem 2. *We have:*

1. *The function $K(n)$, the stabilization length of n, is a computable function of n.*
2. *The function $\|n\|_{st}$, the stable complexity of n, is a computable function of n.*
3. *The set of stable numbers is a computable set.*

It is worth observing here that, strictly speaking, all three parts of this theorem are equivalent. If one has an algorithm for computing $K(n)$, then one may check whether n is stable by checking whether $K(n) = 0$, and one may compute $\|n\|_{st}$ by computing $\|3^{K(n)}n\|$ by the usual methods and observing that

$$\|n\|_{st} = \|3^{K(n)}n\| - 3K(n).$$

Similarly, if one has an algorithm for computing $\|n\|_{st}$, one may compute whether n is stable by checking if $\|n\|_{st} = \|n\|$. Finally, if one has an algorithm for telling if n is stable, one may determine $K(n)$ by simply applying this algorithm to n, $3n$, $9n$, …, until it returns a positive result, which must eventually occur. Such methods for converting between $K(n)$ and $\|n\|_{st}$ may be quite slow, however. Fortunately, the algorithm described here (Algorithm 8) will yield both $K(n)$ and $\|n\|_{st}$ at once, averting such issues; and if one has $K(n)$, checking whether n is stable is a one-step process.

1.3. Applications

An obvious question about $\|n\|$ is that of the complexity of powers, generalizing what was said about powers of 3 above. Certainly for $k \geq 1$ it is true that

$$\|n^k\| \leq k\|n\|,$$

and as noted earlier in the case $n = 3$ we have equality. However other values of n have a more complicated behavior. For instance, powers of 5 do not work nicely, as $\|5^6\| = 29 < 30 = 6 \cdot \|5\|$. The behavior of powers of 2 remains unknown; it has previously been verified [19] that

$$\|2^k\| = k\|2\| = 2k \quad \text{for} \quad 1 \leq k \leq 39.$$

One may combine the known fact that $\|3^k\| = 3k$ for $k \geq 1$, and the hope that $\|2^k\| = 2k$ for $k \geq 1$, into the following conjecture.

Conjecture 1. For $k, \ell \geq 0$ and not both equal to 0,

$$\|2^k 3^\ell\| = 2k + 3\ell.$$

Such a conjecture, if true, is quite far from being proven; after all, it would require that $\|2^k\| = 2k$ for all $k \geq 1$, which would in turn imply that

$$\limsup_{n\to\infty} \frac{\|n\|}{\log n} \geq \frac{2}{\log 2};$$

at present, it is not even known that this limit is any greater than $\frac{3}{\log 3}$, i.e., that $\|n\| \not\sim 3\log_3 n$. Indeed, some have suggested that $\|n\|$ may indeed just be asymptotic to $3\log_3 n$; see [16].

Nonetheless, in this paper we provide some more evidence for this conjecture, by proving the following theorem.

Theorem 3. *For $k \leq 48$ and arbitrary ℓ, so long as k and ℓ are not both zero,*

$$\|2^k 3^\ell\| = 2k + 3\ell.$$

This extends the results of [19] regarding numbers of the form $2^k 3^\ell$, as well as the results of [7], which showed this for $k \leq 21$ and arbitrary ℓ. We prove this not by careful hand analysis, as was done in [7], but by demonstrating, based on the methods of [3], a new algorithm (Algorithm 10) for computing $\|2^k\|$. Not only does it runs much faster than existing algorithms, but it also works, as discussed above, by determining $\|2^k\|_{st}$ and $K(2^k)$, thus telling us whether or not, for the given k, $\|2^k 3^\ell\| = 2k + 3\ell$ holds for all $\ell \geq 0$.

The algorithms here can be used for more purposes as well; see Theorem 9 for a further application of them.

2. Summary of Internals and Further Discussion

2.1. The Defect, Low-defect Polynomials, and Truncation

Let us now turn our attention to the inner workings of these algorithms, which are based on the methods in [3]. Proving the statement $\|n\| = k$ has two parts; showing that $\|n\| \leq k$, and showing that $\|n\| \geq k$. The former is, comparatively, the easy part, as it consists of just finding an expression for n that uses at most k ones; the latter requires ruling out shorter expressions. The simplest method for this is simply exhaustive search, which, as has been mentioned, takes time $\Theta(n^2)$, or time $O(n^{1.231})$ once some possibilities have been eliminated from the addition case.

In this paper, we take a different approach to lower-bounding the quantity $\|n\|$, one used earlier in the paper [7]; however, we make a number of improvements to the method of [7] that both turn this method into an actual algorithm, and frequently allow it to run in a reasonable time. The method is based on considering the *defect* of n, defined below.

Definition 4. The *defect* of n, denoted $\delta(n)$ is defined by

$$\delta(n) := \|n\| - 3\log_3 n.$$

Let us make here a further definition.

Definition 5. For a real number $s \geq 0$, the set A_s is the set of all natural numbers with defect less than s.

The papers [2, 3, 7] provided a method of, for any choice of $\alpha \in (0, 1)$, recursively building up descriptions of the sets $A_\alpha, A_{2\alpha}, A_{3\alpha}, \ldots$; then, if for some n and k we can use this to demonstrate that $n \notin A_{k\alpha}$, then we have determined a lower bound on $\|n\|$. More precisely, they showed that for any $s \geq 0$, there is a finite set $\mathcal{T}_s$ of multilinear polynomials, of a particular form called *low-defect polynomials*, such that $\delta(n) < s$ if and only if n can be written as $f(3^{k_1}, \ldots, 3^{k_r})3^{k_{r+1}}$ for some $f \in \mathcal{T}_s$ and some $k_1, \ldots, k_{r+1} \geq 0$. In this paper, we take this method and show how the polynomials can be produced by an actual algorithm, and how further useful information can be computed once one has these polynomials.

In brief, the algorithm works as follows. First, we choose a step size $\alpha \in (0,1)$. We start with a set of low-defect polynomials representing A_α, and apply the method of [2] to build up sets representing $A_{2\alpha}, A_{3\alpha}, \ldots$; at each step, we use the "truncation" method of [3] to ensure we are representing the set $A_{i\alpha}$ exactly and not including extraneous elements. Then we check whether or not $n \in A_{i\alpha}$; if it is not, we continue on to $A_{(i+1)\alpha}$. If it is, then we have a representation $n = f(3^{k_1}, \ldots, 3^{k_r})3^{k_{r+1}}$, and this gives us an upper bound on $\|n\|$; indeed, we can find a shortest representation for n in this way, and so it gives us $\|n\|$ exactly.

This is, strictly speaking, a little different than what was described above, in that it does not involve directly getting a lower bound on $\|n\|$ from the fact that $n \notin A_{i\alpha}$. However, this can be used too, so long as we know in advance an upper bound on $\|n\|$. For instance, this is quite useful when $n = 2^k$ (for $k \geq 1$), as then we know that $\|n\| \leq 2k$, and hence that $\delta(n) \leq k\delta(2)$. So we can use the method of the above paragraph, but stop early, once we have covered defects up to $k\delta(2) - 1$. If we get a hit within that time, then we have found a shortest representation for $n = 2^k$. Conversely, if n is not detected, then we know that we must have

$$\delta(2^k) > k\delta(2) - 1,$$

and hence that

$$\|2^k\| > 2k - 1,$$

i.e., $\|2^k\| = 2k$, thus verifying that the obvious representation is the best possible. Again, though we have illustrated it here with powers of 2, this method can be used whenever we know in advance an upper bound on $\|n\|$; see the appendix.

Now, so far we have discussed using these methods to compute $\|n\|$, but we can go further and use them to prove Theorem 2, i.e., use them to compute $K(n)$ and $\|n\|_{st}$. In this case, at each step, instead of checking whether there is some $f \in \mathcal{T}_{i\alpha}$ such that $n = f(3^{k_1}, \ldots, 3^{k_r})3^{k_{r+1}}$, we check whether is some $f \in \mathcal{T}_{i\alpha}$ and some ℓ such that

$$3^\ell n = f(3^{k_1}, \ldots, 3^{k_r})3^{k_{r+1}}.$$

It is not immediately obvious that this is possible, since naïvely we would need to check infinitely many ℓ, but Lemma 1 allows us to do this while checking only finitely many ℓ. Once we have such a detection, we can use the value of ℓ to determine $K(n)$, and the representation of $3^\ell n$ obtained this way to determine $\|3^{K(n)}n\|$ and hence $\|n\|_{st}$. In addition, if we know in advance an upper bound on $\|n\|$, we can use the same trick as above to sometimes cut the computation short and conclude not only that $\|n\| = k$ but also that n is stable.

2.2. Comparison to Addition Chains

It is worth discussing some work analogous to this paper in the study of addition chains. An *addition chain* for n is defined to be a sequence $(a_0, a_1, \ldots, a_r)$ such that $a_0 = 1$, $a_r = n$, and, for any $1 \le k \le r$, there exist $0 \le i, j < k$ such that $a_k = a_i + a_j$; the number r is called the length of the addition chain. The shortest length among addition chains for n, called the *addition chain length* of n, is denoted $\ell(n)$. Addition chains were introduced in 1894 by H. Dellac [14] and reintroduced in 1937 by A. Scholz [22]; extensive surveys on the topic can be found in Knuth [20, Section 4.6.3] and Subbarao [24].

The notion of addition chain length has obvious similarities to that of integer complexity; each is a measure of the resources required to build up the number n starting from 1. Both allow the use of addition, but integer complexity supplements this by allowing the use of multiplication, while addition chain length supplements this by allowing the reuse of any number at no additional cost once it has been constructed. Furthermore, both measures are approximately logarithmic; the function $\ell(n)$ satisfies

$$\log_2 n \le \ell(n) \le 2 \log_2 n.$$

A difference worth noting is that $\ell(n)$ is actually known to be asymptotic to $\log_2 n$, as was proved by Brauer [10], but the function $\|n\|$ is not known to be asymptotic to $3 \log_3 n$; the value of the quantity $\limsup_{n \to \infty} \frac{\|n\|}{\log n}$ remains unknown. As mentioned above, Guy [16] has asked whether $\|2^k\| = 2k$ for $k \ge 1$; if true, it would make this quantity at least $\frac{2}{\log 2}$. The Experimental Mathematics Group at the University of Latvia [19] has checked that this is true for $k \le 39$.

Another difference worth noting is that unlike integer complexity, there is no known way to compute addition chain length via dynamic programming. Specifically, to compute integer complexity this way, one may use the fact that for any $n > 1$,

$$\|n\| = \min_{\substack{a,b<n\in\mathbb{N} \\ a+b=n \text{ or } ab=n}} (\|a\| + \|b\|).$$

By contrast, addition chain length seems to be harder to compute. Suppose we have a shortest addition chain $(a_0, \ldots, a_{r-1}, a_r)$ for n; one might hope that $(a_0, \ldots, a_{r-1})$ is a shortest addition chain for a_{r-1}, but this need not be the case. An example is provided by the addition chain $(1, 2, 3, 4, 7)$; this is a shortest addition chain for 7, but $(1, 2, 3, 4)$ is not a shortest addition chain for 4, as $(1, 2, 4)$ is shorter. Moreover, there is no way to assign to each natural number n a shortest addition chain $(a_0, \ldots, a_r)$ for n such that $(a_0, \ldots, a_{r-1})$ is the addition chain assigned to

a_{r-1} [20]. This can be an obstacle both to computing addition chain length and proving statements about addition chains.

Nevertheless, the algorithms described here seem to have a partial analogue for addition chains in the work of A. Flammenkamp [15]. We might define the *addition chain defect* of n by

$$\delta^\ell(n) := \ell(n) - \log_2 n;$$

a closely related quantity, the number of *small steps* of n, was introduced by Knuth [20]. The number of small steps of n is defined by

$$s(n) := \ell(n) - \lfloor \log_2 n \rfloor;$$

clearly, this is related to $\delta^\ell(n)$ by $s(n) = \lceil \delta^\ell(n) \rceil$.

In 1991, A. Flammenkamp determined a method for producing descriptions of all numbers n with $s(n) \le k$ for a given integer k, and produced such descriptions for $k \le 3$ [15]. Note that for k an integer, $s(n) \le k$ if and only if $\delta^\ell(n) \le k$, so this is the same as determining all n with $\delta^\ell(n) \le k$, restricted to the case where k is an integer. Part of what Flammenkamp proved may be summarized as the following theorem.

Theorem 4 (Flammenkamp). *For any integer $k \ge 0$, there exists a finite set $\mathcal{S}_k$ of polynomials (in any number of variables, with nonnegative integer coefficients) such that for any n, one has $s(n) \le k$ if and only if one can write $n = f(2^{m_1}, \ldots, 2^{m_r})2^{m_{r+1}}$ for some $f \in \mathcal{S}_k$ and some integers $m_1, \ldots, m_{r+1} \ge 0$. Moreover, $\mathcal{S}_k$ can be effectively computed.*

Unfortunately, the polynomials used in Flammenkamp's method are more complicated than those produced by the algorithms here; for instance, they cannot always be taken to be multilinear. Nonetheless, there is a distinct similarity.

Flammenkamp did not consider questions of stability (which in this case would result from repeated multiplication by 2 rather than by 3; see [1] for more on this), but it may be possible to use his methods to compute stability information about addition chains, just as the algorithms here may be used to compute stability information about integer complexity. The problem of extending Flammenkamp's methods to allow for non-integer cutoffs seems more difficult.

2.3. Discussion: Algorithms

Many of the algorithms described here are parametric, in that they require a choice of a "step size" $\alpha \in (0, 1)$. In the attached implementation, α is always taken to be $\delta(2) = 0.107\ldots$, and some precomputations have been made based on this choice. See the appendix for more on this. Below, when we discuss the computational complexity of the algorithms given here, we are assuming a fixed choice of α. It is possible that the value of α affects the time complexity of these algorithms. One could also consider what happens when α is considered as an input to the algorithm, so that one cannot do pre-computations based on the choice of α. (In this case we should really restrict the form of α so that the question makes sense, for instance to $\alpha = p - q\log_3 n$, with n a natural number and $p, q \in \mathbb{Q}$.) We will avoid these issues for now, and assume for the rest of this section that $\alpha = \delta(2)$ unless otherwise specified. Two of the algorithms here optionally allow a second input, a known upper bound L on $\|n\|$. If no bound is input, we may think of this as $L = \infty$. We will assume here the simplest case, where no bound L is input, or equivalently where we always pick $L = \infty$.

We will not actually conduct here a formal analysis of the time complexity of Algorithm 8 or Algorithm 10. Our assertion that Algorithm 10 is much faster than existing methods for computing $\|2^k\|$ is an empirical one. The speedup is a dramatic one, though; for instance, the Experimental Mathematics Group's computation of $\|n\|$ for $n \le 10^{12}$ required about 3 weeks on a supercomputer, although they used the $\Theta(n^2)$-time algorithm rather than any of the improvements [18]; whereas computing $\|2^{48}\|$ via Algorithm 10 required only around 20 hours on the author's laptop computer.

Empirically, increasing k by one seems to approximately double the run time of Algorithm 10. This suggests that perhaps Algorithm 10 runs in time $O(2^k)$, which would be better than the $O(2^{1.231k})$ bound coming from applying existing methods [9] to compute the complexity of $\|2^k\|$.

For Algorithm 8, the run time seems to be determined more by the size of $\delta_{st}(n) := \|n\|_{st} - 3\log_3 n$ (or by the size of $\delta(n)$, in the case of Algorithm 9), rather than by the size of n, since it seems that most of the work consists of building the sets of low-defect polynomials, rather than checking if n is represented. For this reason, computing $\|n\|$ via Algorithm 9 is frequently much slower than using existing methods, even though it is much faster for powers of 2. Note that strictly speaking, $\delta(n)$ can be bounded in terms of n, since

$$\delta(n) \le 3\log_2 n - 3\log_3 n,$$

but as mentioned earlier, this may be a substantial overestimate. So it is worth asking the following question.

Question 1. What is the time complexity of Algorithm 10, for computing $K(2^k)$ and $\|2^k\|_{st}$? What is the time complexity of of Algorithm 8 (with $L = \infty$), for computing $K(n)$ and $\|n\|_{st}$? What is the time complexity of Algorithm 9 (with $L = \infty$), for computing the values of $\|3^k n\|$ for a given n and all $k \ge 0$? What if L may be finite? How do these depend on the parameter α? What if α is an input?

2.4. Discussion: Stability and Computation

Although we have now given a means to compute $K(n)$, we have not provided any explicit upper bound on it. The same is true for the quantity

$$\Delta(n) := \|n\| - \|n\|_{st},$$

which is another way of measuring "how unstable" the number n is, and which is also now computable due to Theorem 2. We also do not have any reliable method of generating unstable numbers with which to demonstrate lower bounds.

Empirically, large instabilities – measured either by $K(n)$ or by $\Delta(n)$ – seem to be rare. This statement is not based on running Algorithm 8 on many numbers to determine their stability, as that is quite slow in general, but rather on simply computing $\|n\|$ for $n \le 3^{15}$ and then checking $\|n\|$, $\|3n\|$, $\|9n\|$,..., and guessing that n is stable if no instability is detected before the data runs out, a method that can only ever put lower bounds on $K(n)$ and $\Delta(n)$, never upper bounds. Still, numbers that are detectably unstable at all seem to be somewhat rare, although they still seem to make up a positive fraction of all natural numbers; namely, around 3%. Numbers that are more than merely unstable – having $K(n) \ge 2$ or $\Delta(n) \ge 2$ – are rarer.

The largest lower bounds on $K(n)$ or $\Delta(n)$ for a given n encountered based on these computations occur for $n = 4721323$, which, as mentioned earlier, has $\|3n\| < \|n\|$ and thus $\Delta(n) \ge 4$; and 17 numbers, the smallest of which is $n = 3643$, which have $\|3^5 n\| < \|3^4 n\| + 3$ and thus $K(n) \ge 5$. Finding n where both $K(n)$ and $\Delta(n)$ are decently large is hard; for instance, these computations did not turn up any n for which it could be seen that both $K(n) \ge 3$ and $\Delta(n) \ge 3$.

One may see Table 2.4 for more examples of numbers that seem to have unusual drop patterns. Here, the *drop pattern* of n is the list of values $\delta(3^k n) - \delta(3^{k+1} n)$, or equivalently $\|3^k n\| - \|3^{k+1} n\| + 3$, up until the point where this is always zero. Note that Table 2.4 is empirical, based on the same computation mentioned above; it is possible these numbers have later drops further on. Also note that numbers which are divisible by 3 have been excluded.

It is not even clear whether $K(n)$ or $\Delta(n)$ can get arbitrarily large, or are bounded by some finite constant, although there is no clear reason why the latter would be so. Still, this is worth pointing out as a question.

Question 2. What is the natural density of the set of unstable numbers? What is an explicit upper bound on $K(n)$, or on $\Delta(n)$? Can $K(n)$ and $\Delta(n)$ get arbitrarily large, or are they bounded?

Further questions along these lines suggest themselves, but these questions seem difficult enough, so we will stop this line of inquiry there for now.

Strictly speaking, it is possible to prove Theorem 2 using algorithms based purely on the methods of [2], without actually using the "truncation" method of the paper

Drop pattern	Numbers with this pattern
4	4721323
$1, 2$	1081079
$2, 1$	203999, 1328219
$1, 0, 0, 1$	$153071, 169199$

Table 1: Numbers that seem to have unusual drop patterns

[3]. Of course, one cannot simply remove the truncation step from the algorithms here and get correct answers; other checks are necessary to compensate. See the appendix for a brief discussion of this. However, while this is sufficient to prove Theorem 2, the algorithms obtained this way are simply too slow to be of any use. And without the method of truncation, one cannot write Algorithm 6, without which proving Theorem 9 would be quite difficult. We will demonstrate further applications of Theorem 9 and the method of truncation in future papers [4, 6].

We can also ask about the computational complexity of computing these functions in general, rather than just the specific algorithms here. As noted above, the best known algorithm for computing $\|n\|$ takes time $O(n^{1.231})$. It is also known [8] that the problem "Given n and k in binary, is $\|n\| \le k$?" is in the class NP, because the size of a witness is $O(\log n)$. (This problem is not known to be NP-complete.) However, it is not clear whether the problem "Given n and k in binary, is $\|n\|_{st} \le k$?" is in the class NP, because there is no obvious bound on the size of a witness. It is quite possible that it could be proven to be in NP, however, if an explicit upper bound could be obtained on $K(n)$.

We can also consider the problem of computing the defect ordering, i.e., "Given n_1 and n_2 in binary, is $\delta(n_1) \le \delta(n_2)$?"; the significance of this problem is that the set of all defects is in fact a well-ordered set [2] with order type ω^ω. This problem lies in Δ_2^P in the polynomial hierarchy [2]. The paper [2] also defined the *stable defect* of n.

Definition 6. The *stable defect* of n, denoted $\delta_{st}(n)$, is

$$\delta_{st}(n) := \|n\|_{st} - 3\log_3 n.$$

(We will review the stable defect and its properties in Section 3.1.) Thus we get the problem of, "Given n_1 and n_2 in binary, is $\delta_{st}(n_1) \le \delta_{st}(n_2)$?" The image of δ_{st} is also well-ordered with order type ω^ω, but until now it was not known that this problem is computable. But Theorem 2 shows that it is, and so we can ask about its complexity. Again, due to a lack of bounds on $K(n)$, it is not clear that this lies in Δ_2^P.

We can also ask about the complexity of computing $K(n)$, or $\Delta(n)$ (which, conceivably, could be easier than $\|n\|$ or $\|n\|_{st}$, though this seems unlikely), or, perhaps most importantly, of computing a set $\mathcal{T}_s$ for a given $s \ge 0$. Note that in this last case, it need not be the set $\mathcal{T}_s$ found by Algorithm 6 here; we just want any set satisfying the required properties – a *good covering* of B_s, as we call it here (see Definition 17). Of course, we must make a restriction on the input for this last question, as one cannot actually take arbitrary real numbers as input; perhaps it would be appropriate to restrict to s of the form

$$s \in \{p - q \log_3 n : p, q \in \mathbb{Q}, n \in \mathbb{N}\},$$

which seems like a large enough set of real numbers to cover all the numbers we care about here.

We summarize the above discussion with the following question.

Question 3. What is the complexity of computing $\|n\|$? What is the complexity of computing $\|n\|_{st}$? What is the complexity of computing the difference $\Delta(n)$? What is the complexity of computing the defect ordering $\delta(n_1) \le \delta(n_2)$? What is the complexity of computing the stable defect ordering $\delta_{st}(n_1) \le \delta_{st}(n_2)$? What is the complexity of computing the stabilization length $K(n)$?

Question 4. Given $s = p - q \log_3 n$, with $p, q \in \mathbb{Q}$ and $n \in \mathbb{N}$, what is the complexity of computing a good covering $\mathcal{T}_s$ of B_s?

3. The Defect, Stability, and Low-defect Polynomials

In this section we will review the results of [2] and [3] regarding the defect $\delta(n)$, the stable complexity $\|n\|_{st}$, and low-defect polynomials.

3.1. The Defect and Stability

First, we will need some basic facts about the defect.

Theorem 5. *We have:*

1. *For all n, $\delta(n) \ge 0$.*
2. *For $k \ge 0$, $\delta(3^k n) \le \delta(n)$, with equality if and only if $\|3^k n\| = 3k + \|n\|$. The difference $\delta(n) - \delta(3^k n)$ is a nonnegative integer.*
3. *A number n is stable if and only if for any $k \ge 0$, $\delta(3^k n) = \delta(n)$.*
4. *If the difference $\delta(n) - \delta(m)$ is rational, then $n = m3^k$ for some integer k (and so $\delta(n) - \delta(m) \in \mathbb{Z}$).*
5. *Given any n, there exists k such that $3^k n$ is stable.*

6. *For a given defect α, the set $\{m : \delta(m) = \alpha\}$ has either the form $\{n3^k : 0 \le k \le L\}$ for some n and L, or the form $\{n3^k : 0 \le k\}$ for some n. This latter occurs if and only if α is the smallest defect among $\delta(3^k n)$ for $k \in \mathbb{Z}$.*

7. *$\delta(1) = 1$, and for $k \ge 1$, $\delta(3^k) = 0$. No other integers occur as $\delta(n)$ for any n.*

8. *If $\delta(n) = \delta(m)$ and n is stable, then so is m.*

Proof. Parts (1) through (7), excepting part (3), are just Theorem 2.1 from [2]. Part (3) is Proposition 12 from [7], and part (8) is Proposition 3.1 from [2]. □

The paper [2] also defined the notion of a *stable defect.*

Definition 7. We define a *stable defect* to be the defect of a stable number.

Because of part (9) of Theorem 5, this definition makes sense; a stable defect α is not just one that is the defect of some stable number, but one for which any n with $\delta(n) = \alpha$ is stable. Stable defects can also be characterized by the following proposition from [2].

Proposition 1. *A defect α is stable if and only if it is the smallest defect β such that $\beta \equiv \alpha \pmod 1$.*

We can also define the *stable defect* of a given number, which we denote $\delta_{st}(n)$. (We actually already defined this in Definition 6, but let us disregard that for now and give a different definition; we will see momentarily that they are equivalent.)

Definition 8. For a positive integer n, define the *stable defect* of n, denoted $\delta_{st}(n)$, to be $\delta(3^k n)$ for any k such that $3^k n$ is stable. (This is well-defined as if $3^k n$ and $3^\ell n$ are stable, then $k \ge \ell$ implies $\delta(3^k n) = \delta(3^\ell n)$, and so does $\ell \ge k$.)

Note that the statement "α is a stable defect", which earlier we were thinking of as "$\alpha = \delta(n)$ for some stable n", can also be read as the equivalent statement "$\alpha = \delta_{st}(n)$ for some n".

We then have the following facts relating the notions of $\|n\|$, $\delta(n)$, $\|n\|_{st}$, and $\delta_{st}(n)$.

Proposition 2. *We have:*

1. $\delta_{st}(n) = \min_{k \ge 0} \delta(3^k n)$

2. *$\delta_{st}(n)$ is the smallest defect α such that $\alpha \equiv \delta(n) \pmod 1$.*

3. $\|n\|_{st} = \min_{k \ge 0}(\|3^k n\| - 3k)$

4. $\delta_{st}(n) = \|n\|_{st} - 3 \log_3 n$

5. *$\delta_{st}(n) \le \delta(n)$, with equality if and only if n is stable.*

6. *$\|n\|_{st} \le \|n\|$, with equality if and only if n is stable.*

Proof. These are just Propositions 3.5, 3.7, and 3.8 from [2]. □

3.2. Low-defect Expressions, Polynomials, and Pairs

As has been mentioned in Section 2.1, we are going to represent the set A_r by substituting in powers of 3 into certain multilinear polynomials we call *low-defect polynomials*. Low-defect polynomials come from particular sorts of expressions we will call *low-defect expressions*. We will associate with each polynomial or expression a "base complexity" to form a *low-defect pair*. In this section we will review the properties of these polynomials and expressions.

First, we give a definition.

Definitions 1. A *low defect expression* is defined to be a an expression in positive integer constants, $+$, $\cdot$, and some number of variables, constructed according to the following rules.

1. Any positive integer constant by itself forms a low-defect expression.

2. Given two low-defect expressions using disjoint sets of variables, their product is a low-defect expression. If E_1 and E_2 are low-defect expressions, we will use $E_1 \otimes E_2$ to denote the low-defect expression obtained by first relabeling their variables to be disjoint and then multiplying them.

3. Given a low-defect expression E, a positive integer constant c, and a variable x not used in E, the expression $E \cdot x + c$ is a low-defect expression. (We can write $E \otimes x + c$ if we do not know in advance that x is not used in E.)

We also define an *augmented low-defect expression* to be an expression of the form $E \cdot x$, where E is a low-defect expression and x is a variable not appearing in E. If E is a low-defect expression, we also use $\hat{E}$ to denote the augmented low-defect expression $E \otimes x$.

Note that we do not really care about what variables a low-defect expression is in – if we permute the variables of a low-defect polynomial or replace them with others, we will regard the result as an equivalent low-defect expression.

We also define the complexity of a low-defect expression.

Definitions 2. The *complexity* of a low-defect expression E, denoted $\|E\|$, is the sum of the complexities of all the constants used in E. A *low-defect [expression] pair* is an ordered pair (E, k) where E is a low-defect expression, and k is a whole number with $k \ge \|E\|$.

One can then evaluate these expressions to get polynomials, as given by the following definition.

Definitions 3. A *low-defect polynomial* is a polynomial f obtained by evaluating a low-defect expression E. If (E, k) is a low-defect [expression] pair, we say (f, k) is a low-defect [polynomial] pair. We use $\hat{f}$ to refer to the polynomial obtained by evaluating $\hat{E}$, and call it an *augmented low-defect polynomial.* For convenience, if (f, k) is a low-defect pair, we may say "the degree of (f, k)" to refer to the degree of f.

The reason for introducing the notion of a "low-defect pair" is that we may not always know the complexity of a given low-defect expression; frequently, we will only know an upper bound on it. For more theoretical applications, one does not always need to keep track of this, but since here we are concerned with computation, we need to keep track. One can, of course, always compute the complexity of any low-defect expression one is given; but to do so may be computationally expensive, and it is easier to simply keep track of an upper bound. (Indeed, for certain applications, one may actually want to keep track of more detailed information, such as an upper bound on the complexity of each constant individually; see the appendix for more on this.)

Typically, for practical use, what we want is not either low-defect expressions or low-defect polynomials. Low-defect polynomials do not retain enough information about how they were made. For instance, in the algorithms below, we will frequently want to substitute in values for the "innermost" variables in the polynomial; it is shown in [3] that this is well-defined even if multiple expressions can give rise to the same polynomial. However, if all one has is the polynomial rather than the expression which generated it, determining which variables are innermost may require substantial computation.

On the other hand, low-defect expressions contain unneeded information; there is little practical reason to distinguish between, e.g., $2(3x + 1)$ and $(3x + 1) \cdot 2$, or between $1 \cdot (3x + 1)$ and $3x + 1$, or $2(2(3x + 1))$ and $4(3x + 1)$. A useful practical representation is what [3] called a *low-defect tree*, defined as follows.

Definition 9. Given a low-defect expression E, we define a corresponding *low-defect tree* T, which is a rooted tree where both edges and vertices are labeled with positive integers. We build this tree as follows.

1. If E is a constant n, T consists of a single vertex labeled with n.

2. If $E = E' \cdot x + c$, with T' the tree for E, T consists of T' with a new root attached to the root of T'. The new root is labeled with a 1, and the new edge is labeled with c.

3. If $E = E_1 \cdot E_2$, with T_1 and T_2 the trees for E_1 and E_2 respectively, we construct E by "merging" the roots of E_1 and E_2 – that is to say, we remove

the roots of E_1 and E_2 and add a new root, with edges to all the vertices adjacent to either of the old roots; the new edge labels are equal to the old edge labels. The label of the new root is equal to the product of the labels of the old roots.

See Figure 1 for an example illustrating this construction.

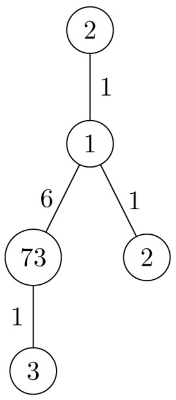

Figure 1: Low-defect tree for the expression $2((73(3x_1+1)x_2+6)(2x_3+1)x_4+1)$.

This will still contain information that is unnecessary for our purposes – for instance, this representation still distinguishes between $4(2x+1)$ and $2(4x+2)$ – but it is on the whole a good medium between including too much and including too little. While the rest of the paper will discuss low-defect expressions and low-defect polynomials, we assume these are being represented as trees, for convenience.

3.3. Properties of Low-defect Polynomials

Having now discussed the definition and representation of low-defect expressions and polynomials, let us now discuss their properties.

Note first that the degree of a low-defect polynomial is also equal to the number of variables it uses; see Proposition 3. We will often refer to the "degree" of a low-defect pair (f, C); this refers to the degree of f. Also note that augmented low-defect polynomials are never low-defect polynomials; as we will see in a moment (Proposition 3), low-defect polynomials always have nonzero constant term, whereas augmented low-defect polynomials always have zero constant term.

Low-defect polynomials are multilinear polynomials; indeed, they are read-once polynomials (in the sense of for instance [25]), as low-defect expressions are easily seen to be read-once expressions.

In [2] the following propositions about low-defect pairs were proved.

Proposition 3. *Suppose f is a low-defect polynomial of degree r. Then f is a polynomial in the variables $x_1, \ldots, x_r$, and it is a multilinear polynomial, i.e., it has degree 1 in each of its variables. The coefficients are non-negative integers. The constant term is nonzero, and so is the coefficient of $x_1 \ldots x_r$, which we will call the* leading coefficient *of f.*

Proposition 4. *If (f, C) is a low-defect pair of degree r, then*

$$\|f(3^{n_1}, \ldots, 3^{n_r})\| \leq C + 3(n_1 + \ldots + n_r).$$

and

$$\|\hat{f}(3^{n_1}, \ldots, 3^{n_{r+1}})\| \leq C + 3(n_1 + \ldots + n_{r+1}).$$

Proof. This is a combination of Proposition 4.5 and Corollary 4.12 from [2]. ☐

Because of this, it makes sense to make the following definition.

Definition 10. Given a low-defect pair (f, C) (say of degree r) and a number N, we will say that (f, C) *efficiently* 3*-represents* N if there exist nonnegative integers $n_1, \ldots, n_r$ such that

$$N = f(3^{n_1}, \ldots, 3^{n_r}) \text{ and } \|N\| = C + 3(n_1 + \ldots + n_r).$$

We will say $(\hat{f}, C)$ efficiently 3-represents N if there exist $n_1, \ldots, n_{r+1}$ such that

$$N = \hat{f}(3^{n_1}, \ldots, 3^{n_{r+1}}) \text{ and } \|N\| = C + 3(n_1 + \ldots + n_{r+1}).$$

More generally, we will also say f 3*-represents* N if there exist nonnegative integers $n_1, \ldots, n_r$ such that $N = f(3^{n_1}, \ldots, 3^{n_r})$, and similarly with $\hat{f}$. We will also use the same terminology regarding low-defect expressions.

Note that if E is a low-defect expression and (E, C) (or $(\hat{E}, C)$) efficiently 3-represents some N, then $(E, \|E\|)$ (respectively, $(\hat{E}, \|E\|)$ efficiently 3-represents N, which means that in order for (E, C) (or $(\hat{E}, C)$) to 3-represent anything efficiently at all, we must have $C = \|E\|$. And if f is a low-defect polynomial and (f, C) (or $\hat{f}, C)$ efficiently 3-represents some N, then C must be equal to the smallest $\|E\|$ among any low-defect expression E that evaluates to f (which in [2] and [3] was denoted $\|f\|$). But, again, it is still worth using low-defect pairs rather than just low-defect polynomials and expressions since we do not want to spend time computing the value $\|E\|$.

For this reason it makes sense to use "E efficiently 3-represents N" to mean "some (E, C) efficiently 3-represents N" or equivalently "$(E, \|E\|)$ efficiently 3-reperesents N". Similarly with $\hat{E}$.

In keeping with the name, numbers 3-represented by low-defect polynomials, or their augmented versions, have bounded defect. We will need some definitions first.

Definition 11. Given a low-defect pair (f, C), we define $\delta(f, C)$, the defect of (f, C), to be $C - 3\log_3 a$, where a is the leading coefficient of f.

Definition 12. Given a low-defect pair (f, C) of degree r, we define

$$\delta_{f,C}(n_1, \ldots, n_r) = C + 3(n_1 + \ldots + n_r) - 3\log_3 f(3^{n_1}, \ldots, 3^{n_r}).$$

Then we obtain the following result.

Proposition 5. *Let (f, C) be a low-defect pair of degree r, and let $n_1, \ldots, n_{r+1}$ be nonnegative integers.*

1. *We have*
$$\delta(\hat{f}(3^{n_1}, \ldots, 3^{n_{r+1}})) \leq \delta_{f,C}(n_1, \ldots, n_r)$$
and the difference is an integer.

2. *We have*
$$\delta_{f,C}(n_1, \ldots, n_r) \leq \delta(f, C)$$
and if $r \geq 1$, this inequality is strict.

3. *The function $\delta_{f,C}$ is strictly increasing in each variable, and*
$$\delta(f, C) = \sup_{k_1, \ldots, k_r} \delta_{f,C}(k_1, \ldots, k_r).$$

Proof. This is a combination of Proposition 4.9 and Corollary 4.14 from [2] along with Proposition 2.14 from [3]. $\square$

Indeed, one can make even stronger statements than (3) above. In [3], a partial order is placed on the variables of a low-defect polynomial f, where, for variables x and y in f, we say $x \preceq y$ if x appears "deeper" in a low-defect expression for f than y does. Formally, we have the following definition.

Definition 13. Let E be a low-defect expression. Let x and y be variables appearing in E. We say that $x \preceq y$ under the *nesting ordering* for E if x appears in the smallest low-defect subexpression of E that contains y.

For instance, if $E = ((((2x_1 + 1)x_2 + 1)(2x_3 + 1)x_4 + 1)x_5 + 1)(2x_6 + 1)$, one has $x_1 \prec x_2 \prec x_4 \prec x_5$ and $x_3 \prec x_4 \prec x_5$ but no other relations. Note that if f is a low-defect polynomial, it can be shown that the nesting order is independent of the low-defect expression used to generate it; see Proposition 3.18 from [3].

With this definition in hand, one can [3, Proposition 4.6] strengthen statement (3) from Proposition 5; for in fact this statement is true even if only the minimal (i.e., innermost) variables are allowed to approach infinity. We now state this more formally.

Proposition 6. *Let (f,C) be a low-defect pair of degree r. Say x_{i_j}, for $1 \le j \le s$, are the minimal variables of f. Then*

$$\lim_{k_{i_1},\ldots,k_{i_s}\to\infty} \delta_{f,C}(k_1,\ldots,k_r) = \delta(f,C)$$

(where the other k_i remain fixed).

Note that if we store the actual low-defect expression rather than just the resulting polynomial, finding the minimal variables is easy.

With this, we have the basic properties of low-defect polynomials.

3.4. Good Coverings

Finally, before we begin listing algorithms, let us state precisely what precisely the algorithms are for. We will first need the notion of a *leader*.

Definition 14. A natural number n is called a *leader* if it is the smallest number with a given defect. By part (6) of Theorem 5, this is equivalent to saying that either $3 \nmid n$, or, if $3 \mid n$, then $\delta(n) < \delta(n/3)$, i.e., $\|n\| < 3 + \|n/3\|$.

We make here another definition as well.

Definition 15. For any real $r \ge 0$, define the set of *r-defect numbers* A_r to be

$$A_r := \{n \in \mathbb{N} : \delta(n) < r\}.$$

Define the set of *r-defect leaders* B_r to be

$$B_r := \{n \in A_r : \quad n \ \text{ is a leader}\}.$$

These sets are related by the following proposition from [2].

Proposition 7. *For every $n \in A_r$, there exists a unique $m \in B_r$ and $k \ge 0$ such that $n = 3^k m$ and $\delta(n) = \delta(m)$; then $\|n\| = \|m\| + 3k$.*

Because of this, if we want to describe the set A_r, it suffices to describe the set B_r.

As mentioned earlier, what we want to do is to be able to write every number in A_r as $f(3^{k_1},\ldots,3^{k_r})3^{k_{r+1}}$ for some low-defect polynomial f drawn from a finite set depending on r. In fact, we want to be able to write every number in B_r as $f(3^{k_1},\ldots,3^{k_r})$, with the same restrictions. Based on this, we make the following definition.

Definition 16. For $r \ge 0$, a finite set $\mathcal{S}$ of low-defect pairs will be called a *covering set* for B_r if every $n \in B_r$ can be efficiently 3-represented by some pair in $\mathcal{S}$. (And hence every $n \in A_r$ can be efficiently represented by some $(\hat{f},C)$ with $(f,C) \in \mathcal{S}$.)

Of course, this is not always enough; we want not just that every number in A_r can be represented in this way, but also that every number generated this way is in A_r. To capture this notion, we make a further definition.

Definition 17. For $r \geq 0$, a finite set $\mathcal{S}$ of low-defect pairs will be called a *good covering* for B_r if every $n \in B_r$ can be efficiently 3-represented by some pair in $\mathcal{S}$ (and hence every $n \in A_r$ can be efficiently represented by some $(\hat{f}, C)$ with $(f, C) \in \mathcal{S}$); and if for every $(f, C) \in \mathcal{S}$, $\delta(f, C) \leq r$, with this being strict if $\deg f = 0$.

With this, it makes sense to state the following theorem from [3].

Theorem 6. *For any real number $r \geq 0$, there exists a good covering of B_r.*

Proof. This is Theorem 4.9 from [3] rewritten in terms of Definition 17. $\square$

Computing good coverings, then, will be one of the primary subjects for the rest of the paper.

Before we continue with that, however, it is also worth noting here the following proposition from [3].

Proposition 8. *Let (f, C) be a low-defect pair of degree k, and suppose that a is the leading coefficient of f. Then $C \geq \|a\| + k$. In particular, $\delta(f, C) \geq \delta(a) + k \geq k$.*

Proof. This is a combination of Proposition 3.24 and Corollary 3.25 from [3]. $\square$

This implies that in any good covering of B_r, all polynomials have degree at most $\lfloor r \rfloor$.

4. Algorithms: Building Up Covering Sets

Now let us discuss the "building-up" method from [7] and [2] that forms one-half the core of the algorithm. The second "filtering-down" half, truncation, will be described in Section 5. This section will describe how to compute covering sets for B_r (see Definition 16); the next section will describe how to turn them into good coverings.

Note however that the results of the above sections and previous papers deal with real numbers, but real numbers cannot be represented exactly in a computer. Hence, we will for the rest of this section fix a subset R of the real numbers on which we can do exact computation. For concreteness, we will define

Definition 18. The set R is the set of all real numbers of the form $q + r \log_3 n$, where q and r are rational and n is a natural number.

This will suffice for our purposes; it contains all the numbers we are working with here. However it is worth noting that all these algorithms will work just as well with a larger set of allowed numbers, so long as it supports all the required operations.

Note that since the algorithms in both this section and later sections consist, in some cases, of simply using the methods described in proofs of theorems in [2] and [3], we will, in these cases, not give detailed proofs of correctness; we will simply direct the reader to the proof of the corresponding theorem. We will include proofs of correctness only where we are not directly following the proof of an earlier theorem.

4.1. Algorithm 1: Computing B_α, $0 < \alpha < 1$

The theorems of [2] that build up covering sets for B_r do so inductively; they require first picking a step size $\alpha \in (0,1)$ and then determining covering sets $B_{k\alpha}$ for natural numbers k. So first, we need a base case – an algorithm to compute B_α. Fortunately, this is given by the following theorem from [7].

Theorem 7. *For every α with $0 < \alpha < 1$, the set of leaders B_α is a finite set. More specifically, the list of n with $\delta(n) < 1$ is as follows:*

1. *3^ℓ for $\ell \geq 1$, of complexity 3ℓ and defect 0*
2. *$2^k 3^\ell$ for $1 \leq k \leq 9$, of complexity $2k + 3\ell$ and defect $k\delta(2)$*
3. *$5 \cdot 2^k 3^\ell$ for $k \leq 3$, of complexity $5 + 2k + 3\ell$ and defect $\delta(5) + k\delta(2)$*
4. *$7 \cdot 2^k 3^\ell$ for $k \leq 2$, of complexity $6 + 2k + 3\ell$ and defect $\delta(7) + k\delta(2)$*
5. *$19 \cdot 3^\ell$ of complexity $9 + 3\ell$ and defect $\delta(19)$*
6. *$13 \cdot 3^\ell$ of complexity $8 + 3\ell$ and defect $\delta(13)$*
7. *$(3^k + 1)3^\ell$ for $k > 0$, of complexity $1 + 3k + 3\ell$ and defect $1 - 3\log_3(1 + 3^{-k})$.*

Strictly speaking, we do not necessarily need this theorem to the same extent as [2] needed it; we only need it if we want to be able to choose step sizes α with α arbitrarily close to 1. In [2], this was necessary to keep small the degrees of the polynomials; larger steps translates into fewer steps, which translates into lower degree. However, in Section 5, we will introduce algorithms for performing truncation, as described in [3]; and with truncation, we can limit the degree without needing large steps (see Corollary 8), allowing us to keep α small if we so choose. For instance, in the attached implementation, we always use $\alpha = \delta(2)$. Nonetheless, one may wish to use larger α, so this proposition is worth noting.

The above theorem can be rephrased as our Algorithm 1, which follows.

```
Algorithm 1 Determine the set $B_\alpha$
Ensure: $\alpha \in (0,1) \cap R$
Require: $T = \{(n,k) : n \in B_\alpha, k = \|n\|\}$
  $T \leftarrow \{(3,3)\}$
  Determine largest integer $k$ such that $k\delta(2) < \alpha$ and $k \leq 9$ {$k$ may be 0, in which case the following loop never executes}
  for $i = 1$ to $k$ do
    $T \leftarrow T \cup \{(2^i, 2i)\}$
  end for
  Determine largest integer $k$ such that $\delta(5) + k\delta(2) < \alpha$ and $k \leq 3$ {$k$ may be negative, in which case the following loop never executes}
  for $i = 0$ to $k$ do
    $T \leftarrow T \cup \{(5 \cdot 2^i, 5 + 2i)\}$
  end for
  Determine largest integer $k$ such that $\delta(7) + k\delta(2) < \alpha$ and $k \leq 2$ {$k$ may be negative, in which case the following loop never executes}
  for $i = 0$ to $k$ do
    $T \leftarrow T \cup \{(7 \cdot 2^i, 6 + 2i)\}$
  end for
  if $\alpha > \delta(19)$ then
    $T \leftarrow T \cup \{(19, 9)\}$
  end if
  if $\alpha > \delta(13)$ then
    $T \leftarrow T \cup \{(13, 8)\}$
  end if
  Determine largest integer $k$ for which $1 - 3\log_3(1 + 3^{-k}) < \alpha$ {$k$ may be 0, in which case the following loop never executes}
  for $i = 1$ to $k$ do
    $T \leftarrow T \cup \{(3^i + 1, 1 + 3i)\}$
  end for
  return $T$
```

Proof of correctness for Algorithm 1. The correctness of this algorithm is immediate from Theorem 7. □

4.2. Algorithm 2: Computing $B_{(k+1)\alpha}$

Now we record Algorithm 2, for computing a covering set for $B_{(k+1)\alpha}$ if we have ones already for $B_\alpha, \ldots, B_{k\alpha}$. This algorithm is essentially the proof of Theorem 4.10 from [2], though we have made a slight modification to avoid redundancy.

Algorithm 2 refers to "solid numbers", and to a set T_α, notions taken from [7], which we have not thus far defined, so let us define those here.

Definitions 4. We say a number n is *solid* if it cannot be efficiently represented as a sum, i.e., there do not exist numbers a and b with $a+b=n$ and $\|a\|+\|b\|=n$. We say a number n is *m-irreducible* if it cannot be efficiently represented as a product, i.e., there do not exist a and b with $ab=n$ and $\|a\|+\|b\|=n$. We define the set T_α to consist of 1 together with those m-irreducible numbers n which satisfy $\frac{1}{n-1} > 3^{\frac{1-\alpha}{3}} - 1$ and do not satisfy $\|n\| = \|n-b\| + \|b\|$ for any solid b with $1 < b \le n/2$.

Algorithm 2 Compute a covering set $\mathcal{S}_{k+1}$ for $B_{(k+1)\alpha}$ from covering sets $\mathcal{S}_1, \ldots, \mathcal{S}_k$ for $B_\alpha, \ldots, B_{k\alpha}$

```
Require: $k \in \mathbb{N}$, $\alpha \in (0,1) \cap R$, $\mathcal{S}_i$ a covering set for $B_{i\alpha}$ for $1 \le i \le k$
Ensure: $\mathcal{S}_{k+1}$ a covering set for $B_{(k+1)\alpha}$
  for all $i = 1$ to $k$ do
    $\mathcal{S}'_i \leftarrow \mathcal{S}_i \setminus \{(1,1),(3,3)\}$
  end for
  $\mathcal{S}_{k+1} \leftarrow \emptyset$
  Compute the set $T_\alpha$, and the complexities of its elements; let $U$ be the set $\{(n, \|n\|) : n \in T_\alpha\}$ {One may use instead a superset of $T_\alpha$ if determining $T_\alpha$ exactly takes too long}
  Compute the set $V_{k,\alpha}$, the set of solid numbers $n$ such that $\|n\| < (k+1)\alpha + 3\log_3 2$ {Again, one may use a superset}
  if $k = 1$ then
    $\mathcal{S}_{k+1} \leftarrow \mathcal{S}_{k+1} \cup \{(f_1 \otimes f_2 \otimes f_3, C_1 + C_2 + C_3) : (f_\ell, C_\ell) \in \mathcal{S}'_1\}$
    $\mathcal{S}_{k+1} \leftarrow \mathcal{S}_{k+1} \cup \{(f_1 \otimes f_2, C_1 + C_2) : (f_\ell, C_\ell) \in \mathcal{S}'_1\}$
  else
    $\mathcal{S}_{k+1} \leftarrow \mathcal{S}_{k+1} \cup \{(f \otimes g, C + D) : (f, C) \in \mathcal{S}'_i, (g, D) \in \mathcal{S}'_j, i + j = k + 2\}$
  end if
  $\mathcal{S}_{k+1} \leftarrow \mathcal{S}_{k+1} \cup \{(f \otimes x + b, C + \|b\|) : (f, C) \in \mathcal{S}_{k\alpha}, b \in V_{k,\alpha}\}$
  $\mathcal{S}_{k+1} \leftarrow \mathcal{S}_{k+1} \cup \{(g \otimes (f \otimes x + b), C + D + \|b\|) : (f, C) \in \mathcal{S}_{k\alpha}, b \in V_{k,\alpha}, (g, D) \in \mathcal{S}'_1\}$
  $\mathcal{S}_{k+1} \leftarrow \mathcal{S}_{k+1} \cup U$
  $\mathcal{S}_{k+1} \leftarrow \mathcal{S}_{k+1} \cup \{(f \otimes g, C + D) : f \in U, g \in \mathcal{S}'_1\}$
  return $\mathcal{S}_{k+1}$
```

Proof of correctness for Algorithm 2. If we examine the proof of Theorem 4.10 from [2], it actually proves the following statement: Suppose that $0 < \alpha < 1$ and that $k \ge 1$. Further suppose that $\mathcal{S}_{1,\alpha}, \mathcal{S}_{2,\alpha}, \ldots, \mathcal{S}_{k,\alpha}$ are covering sets for B_α, $B_{2\alpha}, \ldots, B_{k\alpha}$, respectively. Then we can build a covering set $\mathcal{S}_{k+1,\alpha}$ for $B_{(k+1)\alpha}$ as follows.

1. If $k+1 > 2$, then for $(f, C) \in \mathcal{S}_{i,\alpha}$ and $(g, D) \in \mathcal{S}_{j,\alpha}$ with $2 \le i, j \le k$ and $i + j = k + 2$ we include $(f \otimes g, C + D)$ in $\mathcal{S}_{k+1,\alpha}$; while if $k + 1 = 2$, then for $(f_1, C_1), (f_2, C_2), (f_3, C_3) \in \mathcal{S}_{1,\alpha}$, we include $(f_1 \otimes f_2, C_1 + C_2)$ and $(f_1 \otimes f_2 \otimes f_3, C_1 + C_2 + C_3)$ in $\mathcal{S}_{2,\alpha}$.

2. For $(f, C) \in \mathcal{S}_{k,\alpha}$ and any solid number b with $\|b\| < (k+1)\alpha + 3\log_3 2$, we include $(f \otimes x_1 + b, C + \|b\|)$ in $\mathcal{S}_{k+1,\alpha}$.

3. For $(f, C) \in \mathcal{S}_{k,\alpha}$, any solid number b with $\|b\| < (k+1)\alpha + 3\log_3 2$, and any $v \in B_\alpha$, we include $(v(f \otimes x_1 + b), C + \|b\| + \|v\|)$ in $\mathcal{S}_{k+1,\alpha}$.

4. For all $n \in T_\alpha$, we include $(n, \|n\|)$ in $\mathcal{S}_{k+1,\alpha}$.

5. For all $n \in T_\alpha$ and $v \in B_\alpha$, we include $(vn, \|vn\|)$ in $\mathcal{S}_{k+1,\alpha}$.

Algorithm 2 is, for the most part, exactly this statement. The only difference is the removal of the pairs $(3, 3)$ and $(1, 1)$ from the possibilities of things to multiply by; this step needs additional justification. For $(1, 1)$, this is because no number n can be most-efficiently represented as $1 \cdot n$; if (f, C) is a low-defect pair, then the low-defect pair $(f, C + 1)$ cannot efficiently 3-represent anything, as anything it 3-represents is also 3-represented by the pair (f, C). For $(3, 3)$, there are two possibilities. If $3n$ is a number which is 3-represented by by $(3f, C + 3)$, then either the representation as $3 \cdot n$ is most-efficient or it is not. If it is, then $3n$ is not a leader, and so not in any $B_{i\alpha}$, and thus we do not need it to be 3-represented. If it is not, then it is not efficiently 3-represented by $(3f, C + 3)$. So these particular pairs do not need to be multiplied by, and the algorithm still works. □

4.3. Algorithm 3: Computing a Covering Set for B_r

We can now put the two of these together to form Algorithm 3, for computing a covering set for B_r. If we look ahead to Algorithm 5, we can turn it into a good covering.

Algorithm 3 Compute a covering set for B_r

```
Require: $r \in R, r \geq 0$
Ensure: $S$ is a covering set for $B_r$
  Choose a step size $\alpha \in (0, 1) \cap R$
  Let $T_1$ be the output of Algorithm 1 for $\alpha$ {This is a good covering of $B_\alpha$}
  for $k = 1$ to $\lceil \frac{r}{\alpha} \rceil - 1$ do
    Use Algorithm 2 to compute a covering set $T_{k+1}$ for $B_{(k+1)\alpha}$ from our covering sets
    $T_i$ for $B_{i\alpha}$
    Optional step: Do other things to $T_{k+1}$ that continue to keep it a covering set for
    $B_{(k+1)\alpha}$ while making it more practical to work with. For instance, one may use
    Algorithm 5 to turn it into a good covering of $B_{(k+1)\alpha}$, or one may remove elements
    of $T_{k+1}$ that are redundant (i.e., if one has $(f, C)$ and $(g, D)$ in $T_{k+1}$ such that any
    $n$ which is efficiently 3-represented by $(f, C)$ is also efficiently represented by $(g, D)$,
    one may remove $(f, C)$)
  end for
  $S \leftarrow T_{k+1}$
  return $S$
```

Proof of correctness for Algorithm 3. Assuming the correctness of Algorithm 1 and Algorithm 2, the correctness of Algorithm 3 follows immediately. Again, this is just making use of the proof of Theorem 4.10 from [2]. □

5. Algorithms: Computing Good Coverings

We have now completed the "building-up" half of the method; in this section we will describe the "filtering-down" half. The algorithms here will be based on the proofs of the theorems in [3], so we will once again refer the reader to said proofs in our proofs of correctness.

5.1. Algorithm 4: Truncating a Polynomial to a Given Defect

The first step in being able to filter down is Algorithm 4, for truncating a given polynomial to a given defect.

Algorithm 4 Truncate the low-defect pair (f, C) to the defect s

```
Require: $(f, C)$ is a low-defect pair, $s \in R$
Ensure: $T$ is the truncation of $(f, C)$ to the defect $s$
  if $\deg f = 0$ then
    if $\delta(f, C) < s$ then
      $T \leftarrow \{(f, C)\}$
    else
      $T \leftarrow \emptyset$
    end if
  else
    if $\delta(f, C) \leq s$ then
      $T \leftarrow \{(f, C)\}$
    else
      Find the smallest $K$ for which $\delta_{f,C}(k_1, \ldots, k_r) \geq s$, where $k_i = K + 1$ if $x_i$
      is minimal in the nesting ordering and $x_i = 0$ otherwise
      $T \leftarrow \emptyset$
      for all $x_i$ a minimal variable, $k \leq K$ do
        Let $g$ be $f$ with $3^k$ substituted in for $x_i$ and let $D = C + 3k$
        Recursively apply Algorithm 4 to $(g, D)$ and $s$ to obtain a set $T'$
        $T \leftarrow T \cup T'$
      end for
    end if
  end if
  return $S$
```

Proof of correctness for Algorithm 4. This is an algorithmic version of the method described in the proof of Theorem 4.8 from [3]; see that for details. (Note that K is guaranteed to exist by Proposition 6; one can find it by brute force or slight variants.) There is a slight difference between the two methods in that the method described there, rather than forgetting (f, C) when it recursively applies the method to (g, D) and directly generating the set T, instead generates a set of values for variables that may be substituted into f to yield the set T, only performing the substitution at the end. This is the same method, but without keeping track of extra information so that it can be written in a more straightforwardly recursive manner. □

5.2. Algorithm 5: Truncating Many Polynomials to a Given Defect

If we can truncate one polynomial, we can truncate many of them; this is Algorithm 5.

Algorithm 5 Compute a good covering of B_r from a covering set for B_r

```
Require: $r \in R$, $r \geq 0$, $\mathcal{T}$ a covering set for $B_r$
Ensure: $\mathcal{S}$ is a good covering of $B_r$
  $\mathcal{S} \leftarrow \emptyset$
  for all $(f, C) \in \mathcal{T}$ do
    Use Algorithm 4 to truncate $(f, C)$ to $r$; call the result $\mathcal{S}'$
    $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}'$
  end for
  return $\mathcal{S}$
```

Proof of correctness for Algorithm 5. This is an algorithmic version of the method described in the proof of Theorem 4.9 from [3] – that if one has a covering set for B_r and truncates each of its elements to the defect r, one obtains a good covering of B_r. It can also be seen as an application of the correctness of Algorithms 2 and 4. □

5.3. Algorithm 6: Computing a Good Covering of B_r

We can then put this together into Algorithm 6, for computing a good covering of B_r.

Proof of correctness for Algorithm 6. This follows immediately from the correctness of Algorithms 3 and 5. □

We have now described how to compute good coverings of B_r. But it still remains to show how to use this to compute other quantities of interest, such as $K(n)$ and $\|n\|_{st}$. We address this in the next section.

Algorithm 6 Compute a good covering of B_r

```
Require: $r \in R$, $r \geq 0$
Ensure: $\mathcal{S}$ is a good covering of $B_r$
  Use Algorithm 3 to compute a covering set $\mathcal{T}$ for $B_r$
  Use Algorithm 5 to compute a good covering $\mathcal{S}$ for $B_r$ from $\mathcal{T}$
  return $\mathcal{S}$
```

6. Algorithms: Computing Stabilization Length $K(n)$ and Stable Complexity $\|n\|_{st}$

In order to compute $K(n)$ and $\|n\|_{st}$, we will need to be able to tell, algorithmically, whether, given a low-defect polynomial f and a a number n, there exists $k \geq 0$ such that f 3-represents $3^k n$. If we simply want to know whether f 3-represents n, this is easy; because

$$f(3^{k_1}, \ldots, 3^{k_r}) \geq 3^{k_1+\ldots+k_r},$$

we have an upper bound on how large the k_i can be and we can solve this with brute force. However, if we want to check whether it represents $3^k n$ for any k, clearly this will not suffice, as there are infinitely many possibilities for k. We will need a lemma to narrow them down.

Lemma 1. *Let f be a polynomial in r variables with nonnegative integer coefficients and nonzero constant term; write*

$$f(x_1, \ldots, x_r) = \sum a_{i_1,\ldots,i_r} x_1^{i_1} \ldots x_r^{i_r}$$

with $a_{i_1,\ldots,i_r}$ positive integers and $a_{0,\ldots,0} > 0$. Let $b > 1$ be a natural number and let $v_b(n)$ denote the number of times n is divisible by b. Then for any $k_1, \ldots, k_r \in \mathbb{Z}_{\geq 0}$, we have

$$v_b(f(b^{k_1}, \ldots, b^{k_r})) \leq \sum_{a_{i_1,\ldots,i_r} > 0} (\lfloor \log_b a_{i_1,\ldots,i_r} \rfloor + 1) - 1.$$

In particular, this applies when f is a low-defect polynomial and $b = 3$.

Proof. The number $f(b^{k_1}, \ldots, b^{k_r})$ is the sum of the constant term $a_{0,\ldots,0}$ (call it simply A_0) and numbers of the form $A_i b^{\ell_i}$ where the A_i are simply the remaining $a_{i_1,\ldots,i_r}$ enumerated in some order (say $1 \leq i \leq s$). Since we can choose the order, assume that $v_b(A_1 b^{\ell_1}) \leq \ldots \leq v_b(A_s b^{\ell_s})$.

So consider forming the number $f(b^{k_1}, \ldots, b^{k_r})$ by starting with A_0 and adding in the numbers $A_i b^{\ell_i}$ one at a time. Let S_i denote the sum $\sum_{j=0}^{i} A_j b^{\ell_j}$, so $S_0 = A_0$ and $S_s = f(b^{k_1}, \ldots, b^{k_r})$. We check that for any i, we have

$$v_b(S_i) \leq \sum_{j=0}^{i} (\lfloor \log_b A_j \rfloor + 1) - 1. \qquad (1)$$

Before proceeding further, we observe that if for some i we have $v_b(A_{i+1}b^{\ell_{i+1}}) > v_b(S_i)$, then by assumption, for all $j > i$, $v_b(A_j b^{\ell_j}) \geq v_b(A_{i+1}b^{\ell_{i+1}}) > v_b(S_i)$. Now in general, if $v_b(n) < v_b(m)$, then $v_b(n+m) = v_b(n)$. So we can see by induction that for all $j \geq i$, $v_b(S_j) = v_b(S_i)$; for this statement is true for $j = i$, and if it is true for j, then $v_b(S_j) = v_b(S_i) < v_b(A_j b^{\ell_j})$ and so $v_b(S_{j+1}) = v_b(S_i)$.

So let h be the smallest i such that $v_b(A_{i+1}b^{\ell_{i+1}}) > v_b(S_i)$. (If no such i exists, take $h = s$.) Then we first prove that Equation (1) holds for $i \leq h$.

In the case that $i \leq h$, we will in fact prove the stronger statement that

$$\lfloor \log_b S_i \rfloor \leq \sum_{j=0}^{i} (\lfloor \log_b A_j \rfloor + 1) - 1;$$

this is stronger as in general it is true that $v_b(n) \leq \lfloor \log_b n \rfloor$. For $i = 0$ this is immediate. So suppose that this is true for i and we want to check it for $i+1$, with $i+1 \leq h$. Since $i+1 \leq h$, we have that $v_b(A_{i+1}b^{\ell_{i+1}}) \leq v_b(S_i)$. From this we can conclude the inequality

$$\begin{aligned} \lfloor \log_b(A_{i+1}b^{\ell_{i+1}}) \rfloor &= \ell_{i+1} + \lfloor \log_b A_{i+1} \rfloor \\ \leq v_b(A_{i+1}b^{\ell_{i+1}}) + \lfloor \log_b A_{i+1} \rfloor &\leq v_b(S_i) + \lfloor \log_b A_{i+1} \rfloor. \end{aligned}$$

Now, we also know that

$$\lfloor \log_b S_{i+1} \rfloor \leq \max\{\lfloor \log_b S_i \rfloor, \lfloor \log_b(A_{i+1}b^{\ell_{i+1}}) \rfloor\} + 1. \tag{2}$$

And we can observe using above that

$$\lfloor \log_b(A_{i+1}b^{\ell_{i+1}}) \rfloor + 1 \leq \lfloor \log_b S_i \rfloor + \lfloor \log_b A_{i+1} \rfloor + 1 \leq \sum_{j=0}^{i+1} (\lfloor \log_b A_j \rfloor + 1) - 1.$$

We also know that

$$\lfloor \log_b S_i \rfloor + 1 \leq \sum_{j=0}^{i} (\lfloor \log_b A_j \rfloor + 1) \leq \sum_{j=0}^{i+1} (\lfloor \log_b A_j \rfloor + 1) - 1,$$

as $\lfloor \log_b A_{i+1} \rfloor + 1 \geq 1$. So we can conclude using Equation (2) that

$$\lfloor \log_b S_{i+1} \rfloor \leq \sum_{j=0}^{i+1} (\lfloor \log_b A_j \rfloor + 1) - 1,$$

as desired.

Having proved Equation (1) for $i \leq h$, it then immediately follows for all i, as by the above, for $i \geq h$,

$$v_b(S_i) = v_b(S_h) \leq \sum_{j=0}^{h} (\lfloor \log_b A_j \rfloor + 1) - 1 \leq \sum_{j=0}^{s} (\lfloor \log_b A_j \rfloor + 1) - 1;$$

this proves the claim. □

6.1. Algorithm 7: Computing Whether a Polynomial 3-represents Some $3^k n$.

With this in hand, we can now write down Algorithm 7 for determining if f 3-represents any $3^k n$.

Algorithm 7 Determine whether (f, C) 3-represents any $3^k n$ and with what complexities

```
Require: $(f, C)$ a low-defect pair, $n$ a natural number
Ensure: $S$ is the set of $(k, \ell)$ such that there exist whole numbers $(k_1, \ldots, k_r)$ with
  $f(3^{k_1}, \ldots, 3^{k_r}) = 3^k n$ and $C + 3(k_1 + \ldots + k_r) = \ell$
  $S \leftarrow \emptyset$
  Determine $v$ such that for any $k_1, \ldots, k_r$, one has $v_3(f(3^{k_1}, \ldots, 3^{k_r})) \leq v$ {one
  method is given by Lemma 1}
  for $k = 0$ to $v - v_3(n)$ do
    for all $(k_1, \ldots, k_r)$ such that $k_1 + \ldots + k_r \leq k + \lfloor \log_3 n \rfloor$ do
      if $f(3^{k_1}, \ldots, 3^{k_r}) = 3^k n$ then
        $S \leftarrow S \cup \{(k, C + 3(k_1 + \ldots + k_r))\}$
      end if
    end for
  end for
  return $S$
```

Proof of correctness for Algorithm 7. Once we have picked a v (which can be found using Lemma 1), it suffices to check if f represents $3^k n$ with $k + v_3(n) \leq v$. By Proposition 3, for any $k_1, \ldots, k_r$, we have

$$f(3^{k_1}, \ldots, 3^{k_r}) \geq 3^{k_1 + \ldots + k_r},$$

and so it suffices to check it for tuples $(k_1, \ldots, k_r)$ with $k_1 + \ldots + k_r \leq \lfloor \log_3 3^k n \rfloor$. There are only finitely many of these and so this can be done by brute force, and this is exactly what the algorithm does. □

Note that Algorithm 7 is for determining specifically if there is some $k \geq 0$ such that f 3-represents $3^k n$; it is not for $k \leq 0$. In order to complete the algorithms that follow, we will also need to be able to check if there is some $k \leq 0$ such that f 3-represents $3^k n$. However, this is the same as just checking if $\hat{f}$ 3-represents n, and can be done by the same brute-force methods as were used to check if f 3-represents n; no special algorithm is required here.

6.2. Algorithm 8: Algorithm to Test Stability and Compute Stable Complexity

Now, at last, we can write down Algorithm 8, for computing $K(n)$ and $\|n\|_{st}$. We assume that in addition to n, we are given L, an upper bound on $\|n\|$, which may be ∞. Running Algorithm 8 with $L = \infty$ is always a valid choice; alternatively, one may compute $\|n\|$ or an upper bound on it before applying Algorithm 8.

Algorithm 8 Compute $K(n)$ and $\|n\|_{st}$

```
Require: $n$ a natural number, $L \in \mathbb{N} \cup \{\infty\}$, $L \geq \|n\|$
Ensure: $(k, m) = (K(n), \|n\|_{st})$
  Choose a step size $\alpha \in (0, 1) \cap R$
  Let $r$ be the smallest nonnegative integer, or $\infty$, such that $r\alpha > L - 3\log_3 n - 1$
  $i \leftarrow 1$
  $U \leftarrow \emptyset$
  while $U = \emptyset$ and $i \leq r$ do
    if $i = 1$ then
      Let $\mathcal{S}_1$ be the output of Algorithm 1 for $\alpha$ {This is a good covering of $B_\alpha$}
    else
      Use Algorithm 2 to compute a covering $\mathcal{S}_i$ of $B_{i\alpha}$ from coverings $\mathcal{S}_j$ of $B_{j\alpha}$
      for $1 \leq j < i$
      Use Algorithm 5 to turn $\mathcal{S}_i$ into a good covering
    end if
    Optional step: Remove redundancies from $\mathcal{S}_i$ as in Algorithm 2 {See "optional
    step" there}
    for all $(f, C) \in \mathcal{S}_i$ do
      Let $U'$ be the output of Algorithm 7 on $(f, C)$ and $n$ {If $r$ is finite and $i < r$
      this whole loop may be skipped}
      Let $s = \deg f$
      for all $(k_1, \ldots, k_{s+1})$ such that $k_1 + \ldots + k_{s+1} \leq \lfloor \log_3 n \rfloor$ do
        if $\hat{f}(3^{k_1}, \ldots, 3^{k_{s+1}}) = n$ then
          $U' \leftarrow U' \cup \{(k, C + 3(k_1 + \ldots + k_{s+1}))\}$
        end if
      end for
      $U \leftarrow U \cup U'$
    end for
  end while
  if $U = \emptyset$ then
    $(k, m) = (0, L)$
  else
    Let $V$ consist of the elements $(k, \ell)$ of $U$ that minimize $\ell - 3k$
    Choose $(k, \ell) \in V$ that minimizes $k$
    $m \leftarrow \ell - 3k$
  end if
  return $(k, m)$
```

Proof of correctness for Algorithm 8. This algorithm progressively builds up good covers $\mathcal{S}_i$ of $B_{i\alpha}$ until it finds some i such that there is some $(f, C) \in \mathcal{S}_i$ such that $\hat{f}$

3-represents $3^k n$ for some $k \geq 0$. To see that this is indeed what it is doing, observe that if

$$f(3^{k_1}, \ldots, 3^{k_r})3^{k_{r+1}} = 3^k n,$$

then if $k \geq k_{r+1}$, we may write

$$f(3^{k_1}, \ldots, 3^{k_r}) = 3^{k-k_{r+1}} n$$

and so f itself 3-represents some $3^k n$; while if $k \leq k_{r+1}$, we may write

$$f(3^{k_1}, \ldots, 3^{k_r})3^{k_{r+1}-k} = n$$

and so $\hat{f}$ 3-represents n itself. And this is exactly what the inner loop does; it checks if f 3-represents any $3^k n$ using Algorithm 7, and it checks if $\hat{f}$ 3-represents n using brute force.

Now, if for a given i we obtain $U = \emptyset$, then that means that no $3^k n$ is 3-represented by any $(f, C) \in \mathcal{S}_i$, and so for any k, $\delta(3^k n) \geq i\alpha$, that is, $\delta_{st}(3^k n) \geq i\alpha$. Conversely, if for a given i we obtain U nonempty, then that means that some $3^k n$ is 3-represented by some $(f, C) \in \mathcal{S}_i$. Since for any (f, C) we have $\delta(f, C) \leq i\alpha$ (and this is strict if $\deg f = 0$), this means that $\delta(3^k n) < i\alpha$, and so $\delta_{st}(n) < i\alpha$.

So we see that if the algorithm exits the main loop with U nonempty, it does so once it has found some i such that there exists k with $\delta(3^k n) < i\alpha$; equivalently, once it has found some i such that $\delta_{st}(n) < i\alpha$. Or, equivalently, once it has found some i such that $\delta(3^{K(n)} n) < i\alpha$. Furthermore, note that $3^{K(n)} n$ must be a leader if $K(n) > 0$, as otherwise $3^{K(n)-1} n$ would also be stable. So if $K(n) > 0$, then $3^{K(n)} n$ must be efficiently 3-represented by some $(f, C) \in \mathcal{S}_i$. Whereas if $K(n) = 0$, then we only know that it is efficiently 3-represented by some $(\hat{f}, C)$ for some $(f, C) \in \mathcal{S}_i$, but we also know $3^{K(n)} n = n$. That is to say, the ordered pair $(K(n), \|3^{K(n)} n\|)$ must be in the set U.

In this case, where U is nonempty, it remains to examine the set U and pick out the correct candidate. Each pair $(k, \ell) \in U$ consists of some k and some ℓ such that $\ell \geq \|3^k n\|$. This implies that

$$\delta_{st}(n) \leq \delta(3^k n) \leq \ell - 3k - 3\log_3 n,$$

and so the pair $(K(n), \|3^{K(n)} n\|)$ must be a pair (k, ℓ) for which the quantity $\ell - 3k - 3\log_3 n$, and hence the quantity $\ell - 3k$, is minimized; call this latter minimum p. So

$$\delta_{st}(n) = p - 3\log_3 n.$$

(Note that this means that $p = \|n\|_{st}$.) Then the elements of V are pairs $(k, p + 3k)$ with

$$\delta(3^k n) \leq p - 3\log_3 n,$$

but we know also that

$$\delta(3^k n) \geq \delta_{st}(n) = p - 3\log_3 n,$$

so we conclude that for such a pair, $\delta(3^k n) = \delta_{st}(n)$. But this means that $3^k n$ is stable, and so $k \geq K(n)$. But we know that $K(n)$ is among the set of k with $(k, p+3k) \in V$, and so it is their minimum. Thus, we can select the element $(k, \ell) \in V$ that minimizes k; then $k = K(n)$, and we can take $k - 3\ell$ to find $m = \|n\|_{st}$.

This leaves the case where U is empty. In this case, we must have that for all $1 \leq i \leq r$, and hence in particular for $i = r$, no (f, C) in $\mathcal{S}_i$ 3-represents any $n3^k$; i.e., no $n3^k$ lies in $B_{r\alpha}$, and hence, by Proposition 7, no $n3^k$ lies in $A_{r\alpha}$. That is to say, for any k, $\delta(n3^k) \geq r\alpha$, and so

$$\|n3^k\| \geq r\alpha + 3\log_3 n + 3k > L + 3k - 1.$$

Since $\|n3^k\| > L + 3k - 1$, and $\|n3^k\| \leq L + 3k$, we must have $\|n3^k\| = L + 3k$. Since this is true for all $k \geq 0$, we can conclude that n is a stable number. So, n is stable and $\|n\| = L$, that is to say, $K(n) = 0$ and $\|n\|_{st} = \|n\| = L$. $\square$

We can now prove Theorem 2.

Proof of Theorem 2. Algorithm 8, run with $L = \infty$, gives us a way of computing $K(n)$ and $\|n\|_{st}$. Then, to check if n is stable, it suffices to check whether or not $K(n) = 0$. This proves the theorem. $\square$

6.3. Algorithm 9: Determining Leaders and the "Drop Pattern"

But we need not conclude here; we can go further. As mentioned in Section 2.1, we can get more information if we go until we detect n, rather than stopping as soon as we detect some $3^k n$. We now record Algorithm 9, for not only determining $K(n)$ and $\|3^{K(n)}n\|$, but for determining all k such that either $k = 0$ or $3^k n$ is a leader, and the complexities $\|3^k n\|$. By Proposition 7, this is enough to determine $\|3^k n\|$ for all $k \geq 0$. One could also do this by using Algorithm 8 to determine $K(n)$ and then directly computing $\|3^k n\|$ for all $0 \leq k \leq K(n)$, but Algorithm 9 will often be faster.

Proof of correctness for Algorithm 9. As in Algorithm 8, we are successively building up good coverings $\mathcal{S}_i$ of $B_{i\alpha}$, and for each one checking whether there is an $(f, C) \in \mathcal{S}_i$ and a $k \geq 0$ such that $(\hat{f}, C)$ 3-represents $3^k n$. However, the exit condition on the loop is different; ignoring for a moment the possibility of exiting due to $i > r$, the difference is that instead of stopping once some $3^k n$ is 3-represented, we do not stop until n itself is 3-represented, or equivalently, $\delta(n) < i\alpha$. We will use i here to denote the value of i when the loop exits.

Algorithm 9 Compute information determining $\|3^k n\|$ for all $k \geq 0$

```
Require: $n$ a natural number, $L \in \mathbb{N} \cup \{\infty\}$, $L \geq \|n\|$
Ensure: $V$ the set of $(k, \ell)$ where either $k = 0$ or $k > 0$ and $3^k n$ is a leader, and
  $\ell = \|3^k n\|$
  Choose a step size $\alpha \in (0,1) \cap R$
  Let $r$ be the smallest nonnegative integer, or $\infty$, such that $r\alpha > L - 3\log_3 n - 1$
  $i \leftarrow 1$
  $U \leftarrow \emptyset$
  while $0 \notin \pi_1(U)$, where $\pi_1$ is projection onto the first coordinate, and $i \leq r$ do
    if $i = 1$ then
      Let $\mathcal{S}_1$ be the output of Algorithm 1 for $\alpha$ {This is a good covering of $B_\alpha$}
    else
      Use Algorithm 2 to compute a covering $\mathcal{S}_i$ of $B_{i\alpha}$ from coverings $\mathcal{S}_j$ of $B_{j\alpha}$
      for $1 \leq j < i$
      Use Algorithm 5 to turn $\mathcal{S}_i$ into a good covering
    end if
    Optional step: Remove redundancies from $\mathcal{S}_i$ as in Algorithm 2 {See "optional
    step" there}
    for all $(f, C) \in \mathcal{S}_i$ do
      Determine $v$ such that for any $k_1, \ldots, k_r$, one has $v_3(f(3^{k_1}, \ldots, 3^{k_r})) \leq v$
      {one method is given by Lemma 1} {If $r$ is finite and $i < r$ this whole loop
      may be skipped}
      Let $U'$ be the output of Algorithm 7 on $(f, C)$ and $n$
      for all $(k_1, \ldots, k_{r+1})$ such that $k_1 + \ldots + k_{r+1} \leq \lfloor \log_3 n \rfloor$ do
        if $\hat{f}(3^{k_1}, \ldots, 3^{k_{r+1}}) = n$ then
          $U' \leftarrow U' \cup \{(k, C + 3(k_1 + \ldots + k_{r+1}))\}$
        end if
      end for
      $U \leftarrow U \cup U'$
    end for
  end while
  if $0 \notin \pi_1(U)$ then
    $U \leftarrow U \cup \{(0, L)\}$
  end if
  Let $V = \{(k, \ell - 3k) : (k, \ell) \in U\}$
  Let $V_m$ consist of the minimal elements of $V$ in the usual partial order
  Let $W = \{(k, p + 3k) : (k, p) \in V_m\}$
  return $W$
```

We want the set U to have two properties. Firstly, it should contain all the pairs (k, ℓ) we want to find. Secondly, for any $(k, \ell) \in U$, we should have $\|3^k n\| \leq \ell$. For the first property, observe that if $3^k n$ is a leader and $k > 1$, then

$$\delta(3^k n) \le \delta(n) - 1 < L - 3\log_3 n - 1,$$

and so $\delta(3^k n) \le r\alpha$; thus, $3^k n$ (being a leader) is efficiently 3-represented by some $(f, C) \in \mathcal{S}_r$, and so if the loop exits due to $i > r$, then $(k, \|3^k n\|) \in U$. Whereas if the loop exits due to $0 \in \pi_1(U)$, then note $\delta(3^k n) \le \delta(n) < i\alpha$, and so $3^k n$ (again being a leader) is efficiently 3-represented by some $(f, C) \in S_i$, and so again $(k, \|3^k n\|) \in U$. This leaves the case where $k = 0$. If the loop exits due to $0 \in \pi_1(U)$, then by choice of i, n is efficiently 3-represented by some $(\hat{f}, C)$ for some $(f, C) \in \mathcal{S}_i$, so $(0, \|n\|) \in U$. Whereas if the loop exits due to $i > r$, then this means that $\delta(n) \ge r\alpha$, and so

$$\|n\| \ge r\alpha + 3\log_3 n > L - 1;$$

since we know $\|n\| \le L$, this implies $\|n\| = L$, and so including $(0, L)$ in U means $(0, \|n\|) \in U$.

For the second property, again, there are two ways a pair (k, ℓ) may end up in U. One is that some low-defect pair (f, C) 3-represents the number $3^k n$, which, as in the proof of correctness for Algorithm 8, means $\|3^k n\| \le \ell$. The other is that $(k, \ell) = (0, L)$; but in this case, $\|n\| \le L$ by assumption.

It then remains to isolate the pairs we want from the rest of U. We will show that they are in fact precisely the minimal elements of U under the partial order

$$(k_1, \ell_1) \le (k_2, \ell_2) \iff k_1 \le k_2 \text{ and } \ell_1 - 3k_1 \le \ell_2 - 3k_2.$$

Say first that (k, ℓ) is one of the pairs we are looking for, i.e, either $k = 0$ or $3^k n$ is a leader, and $\ell = \|3^k n\|$. Now suppose that that $(k', \ell') \in U$ such that $k' \le k$ and $\ell - 3k' \le \ell - 3k$. Since $(k', \ell') \in U$, that means that $\|3^{k'} n\| \le \ell'$. Since $k' \le k$, we conclude that

$$\ell = \|3^k n\| \le \ell' + 3(k - k') \tag{3}$$

and hence that $\ell - 3k \le \ell' - 3k'$, so $\ell - 3k = \ell' - 3k'$. Now, if $k = 0$, then certainly $k \le k'$ (and so $k = k'$); otherwise, $3^k n$ is a leader. Suppose we had $k' < k$; then since $3^k n$ is a leader, that would mean $\delta(3^k n) < \delta(3^{k'} n)$ and hence

$$\|3^k n\| < \|3^{k'} n\| + 3(k - k') = \ell + 3(k - k'),$$

contrary to (3). So we conclude $k' = k$, and so (k, ℓ) is indeed minimal.

Conversely, suppose that (k, ℓ) is a minimal element of U in this partial order. We must show that $\ell = \|3^k n\|$, and, if $k > 0$, that $3^k n$ is a leader. Choose $k' \le k$ as large as possible with either $k' = 0$ or $3^{k'} n$ a leader, so that $\delta(3^{k'} n) = \delta(3^k n)$. Also, let $\ell' = \|3^{k'} n\|$; by above, $(k', \ell') \in U$. Since $(k, \ell) \in U$ and $\delta(3^{k'} n) = \delta(3^k n)$, we know that

$$\|3^{k'} n\| + 3(k - k') = \|3^k n\| \le \ell$$

and hence $\ell' - 3k' \le \ell - 3k$. Since by assumption we also have $k' \le k$, by the assumption of minimality we must have $(k', \ell') = (k, \ell)$. But this means exactly that either $k = 0$ or $3^k n$ is a leader, and that

$$\|3^k n\| = \|3^{k'} n\| = \ell' = \ell,$$

as needed. □

6.4. Algorithm 10: Stabilization Length and Stable Complexity for $n = 2^k$

Finally, before moving on to the results of applying these algorithms, we make note of one particular specialization of Algorithm 8, namely, the case where $n = 2^k$ and $\ell = 2k$. As was noted in Section 2.3, this turns out to be surprisingly fast as a method of computing $\|2^k\|$. We formalize it here.

Algorithm 10 Given $k \ge 1$, determine $K(2^k)$ and $\|2^k\|_{st}$

```
Require: $k \ge 1$ an integer
Ensure: $(h,p) = (K(2^k), \|2^k\|_{st})$
  Let $(h,p)$ be the result of applying Algorithm 8 with $n = 2^k$ and $L = 2k$.
  return $(h,p)$
```

Proof of correctness for Algorithm 10. This follows from the correctness of Algorithm 8 and the fact that $\|2^k\| \le 2k$ for $k \ge 1$. □

7. Further Notes on Stabilization and Stable Complexity

Before we continue on to the results of applying these algorithms, let us make a few more notes on the stabilization length $K(n)$ and the stable complexity $\|n\|_{st}$, now that we have demonstrated how to compute them. We begin with the following inequality.

Proposition 9. *For natural numbers n_1 and n_2, $\|n_1 n_2\|_{st} \le \|n_1\|_{st} + \|n_2\|_{st}$.*

Proof. Choose k_1, k_2, and K such that $k_1 + k_2 = K$, both $3^{k_i} n_i$ are stable, and $3^K n_1 n_2$ is also stable. Then

$$\|n_1 n_2\|_{st} = \|3^K n_1 n_2\| - 3K \le \|3^{k_1} n_1\| + \|3^{k_2} n_2\| - 3(k_1 + k_2) = \|n_1\|_{st} + \|n_2\|_{st}.$$

□

Unfortunately, the analogous inequality for addition does not hold; for instance,

$$\|2\|_{st} = 2 > 0 = \|1\|_{st} + \|1\|_{st};$$

more examples can easily be found.

As was mentioned in Section 2.4, we can measure the instability of the number n by the quantity $\Delta(n)$, defined as

$$\Delta(n) = \|n\| - \|n\|_{st} = \delta(n) - \delta_{st}(n).$$

We can also measure of how far from optimal a factorization is – and, due to Proposition 9, we can do a stabilized version of this as well.

Definitions 5. Let $n_1, \ldots, n_k$ be positive integers, and let N be their product. We define $\kappa(n_1, \ldots, n_k)$ to be the difference $\|n_1\| + \ldots + \|n_r\| - \|N\|$. Similarly we define $\kappa_{st}(n_1, \ldots, n_k)$ to be the difference $\|n_1\|_{st} + \ldots + \|n_k\|_{st} - \|N\|_{st}$.

If $\kappa(n_1, \ldots, n_k) = 0$, we will say that the factorization $N = n_1 \cdots n_k$ is a *good factorization*. If $\kappa_{st}(n_1, \ldots, n_k) = 0$, we will say that the factorization $N = n_1 \cdots n_k$ is a *stably good factorization*.

These definitions lead to the following easily-proved but useful equation.

Proposition 10. *Let $n_1, \ldots, n_k$ be natural numbers with product N. Then*

$$\Delta(N) + \kappa(n_1, \ldots, n_k) = \sum_{i=1}^{k} \Delta(n_i) + \kappa_{st}(n_1, \ldots, n_k).$$

Proof. Both sides are equal to the difference $\sum_{i=1}^{k} \|n_i\| - \|N\|_{st}$. □

The usefulness of this equation comes from the fact that all the summands are nonnegative integers. For instance, we can obtain from it the following implications.

Corollary 1. *Let $n_1, \ldots, n_k$ be natural numbers with product N; consider the factorization $N = n_1 \cdot \ldots \cdot n_k$. Then:*

1. *If N is stable and the factorization is good, then the n_i are stable.*
2. *If the n_i are stable and the factorization is stably good, then N is stable.*
3. *If the factorization is stably good, then $K(N) \leq \sum_i K(n_i)$.*

(Part (1) of this proposition also appeared as Proposition 24 in [7].)

Proof. For part (1), by Proposition 10, if $\Delta(N) = \kappa(n_1, \ldots, n_k) = 0$, then we must have that $\Delta(n_i) = 0$ for all i, i.e., the n_i are all stable. For part (2), again by Proposition 10, if $\kappa_{st}(n_1, \ldots, n_k) = 0$ and $\Delta(n_i) = 0$ for all i, then we must have $\Delta(N) = 0$, i.e., N is stable. Finally, for part (3) let $K_i = K(N_i)$, and let $K = K_1 + \ldots + K_r$. Then $\prod_i (3^{K_i} n_i) = 3^K n$. Now by hypothesis,

$$\kappa_{st}(3^{K_1} n_1, \ldots, 3^{K_r} n_r) = \kappa_{st}(n_1, \ldots, n_r) = 0,$$

and furthermore each $3^{K_i} n_i$ is stable. Hence by part (2), we must also have that $3^K N$ is stable, that is, that $K(N) \leq K = K(N_1) + \ldots + K(N_r)$. □

Having noted this, let us now continue on towards the results of actually performing computations with these algorithms.

8. Results of Computation

Armed with our suite of algorithms, we now proceed to the results of our computations. First, we can use Algorithm 10 to prove Theorem 3.

Proof of Theorem 3. Algorithm 10 was applied with $k = 48$, and it was determined that $K(2^{48}) = 0$ and $\|2^{48}\|_{st} = 96$, that is to say, that 2^{48} is stable and $\|2^{48}\| = 96$, that is to say, that $\|2^{48} 3^\ell\| = 96 + 3\ell$ for all $\ell \geq 0$. This implies that $\|2^k 3^\ell\| = 2k + 3\ell$ for all $0 \leq k \leq 48$ and $\ell \geq 0$ with k and ℓ not both zero, as if one instead had $\|2^k 3^\ell\| < 2k + 3\ell$, then writing $2^{48} 3^\ell = 2^{48-k}(2^k 3^\ell)$, one would obtain $2^{48} 3^\ell < 96 + 3\ell$. □

But we can do more with these algorithms than just straightforward computation of values of complexities and stable complexities. For instance, we can answer the question, what is the smallest unstable defect other than 1?

In [7], the following theorem was proven.

Theorem 8. *For any $n > 1$, if $\delta(n) < 12\delta(2)$, then n is stable.*

That is to say, with the exception of 1, all defects less than $12\delta(2)$ are stable. This naturally leads to the question, what is the smallest unstable defect (other than 1)? We might also ask, what is the smallest unstable number (other than 1)? Interestingly, among unstable numbers greater than 1, the number 107 turns out to be smallest both by magnitude and by defect. However, if we measure unstable numbers (other than 1) by their stable defect, the smallest will instead turn out to be 683. We record this in the following theorem.

Theorem 9. *We have:*

1. *The number 107 is the smallest unstable number other than 1.*

2. *Other than 1, the number 107 is the unstable number with the smallest defect, and $\delta(107) = 3.2398\ldots$ is the smallest unstable defect other than 1.*

3. *Among nonzero values of $\delta_{st}(n)$ for unstable n, the defect $\delta_{st}(683)$, or equivalently $\delta(2049) = 2.17798\ldots$, is the smallest.*

Proof. For part (1), it suffices to use Algorithm 8 to check the stability of all numbers from 2 to 106.

For parts (2) and (3), in order to find unstable numbers of small defect, we will search for leaders of small defect which are divisible by 3. (Since if n is unstable, then $3^{K(n)}n$ is a leader divisible by 3, and $\delta(3^{K(n)}n) < \delta(n)$). We use Algorithm 6 to compute a good covering S of $B_{21\delta(2)}$. Doing a careful examination of the low-defect polynomials that appear, we can determine all the multiples of 3 that each one can 3-represent; we omit this computation, but its results are that the following multiples of 3 can be 3-represented by: 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 54, 57, 60, 63, 66, 72, 75, 78, 81, 84, 90, 96, 111, 114, 120, 126, 129, 132, 144, 162, 165, 168, 171, 180, 192, 225, 228, 231, 240, 252, 258, 264, 288, 321, 324, 330, 336, 360, 384, 480, 513, 516, 528, 576, 768, 1026, 1032, 1056, 1152, 1536, 2049, 2052, 2064, 2112, 2304, 3072, and, for $k \geq 0$, numbers of the forms $12 \cdot 3^k + 3$, $6 \cdot 3^k + 3$, $9 \cdot 3^k + 3$, $12 \cdot 3^k + 6$, and $18 \cdot 3^k + 6$.

For the individual leaders, we can easily check by computation that the only ones which are leaders are 3, 321, and 2049. This leaves the infinite families. For these, observe that if we divide them by 3, we get, respectively, $4 \cdot 3^k + 1$, $2 \cdot 3^k + 1$, $3 \cdot 3^k + 1$, $2(2 \cdot 3^k + 1)$, and $2(3 \cdot 3^k + 1)$, and it is easy to check that any number of any of those forms has defect less than $12\delta(2)$ and hence is stable by Theorem 8; thus, multiplying them by 3 cannot yield a leader.

So we conclude that the only leaders m with $\delta(m) < 21\delta(2)$ are 3, 321, and 2049. Therefore, the only unstable numbers n with $\delta_{st}(n) < 21\delta(2)$ are 1, 107, and 683. Note also that by the above computation, no power of 3 times any of 3, 321, or 2049 is a leader (as it would have to have smaller defect and would thus appear in the list), and thus the numbers 3, 321, and 2049 are not just leaders but in fact stable leaders. So to prove part (3), it suffices to note that, since $\delta_{st}(3) = 0$, among $\delta_{st}(107)$ (i.e. $\delta(321)$) and $\delta_{st}(683)$ (i.e. $\delta(2049)$), the latter is smaller.

This leaves part (2). Observe that $\delta(107) = \delta(321)+1$. And if n is unstable, then $\delta_{st}(n) \leq \delta(n) - 1$. So if $n > 1$ is unstable and $\delta(n) < \delta(107)$, then $\delta_{st}(n) < \delta(321)$, which by the above forces $n = 683$. But in fact, although $\delta(2049) < \delta(107)$, we nonetheless have $\delta(683) > \delta(107)$ (because while $\delta(107) = \delta(321) + 1$, $\delta(683) = \delta(2049) + 2$). Thus $\delta(107)$ is the smallest unstable defect other than 1, i.e., 107 is (other than 1) the smallest unstable number by defect. □

These computational results provide a good demonstration of the power of the methods here.

Acknowledgements. The author is grateful to J. Arias de Reyna for helpful discussion. He thanks his advisor J. C. Lagarias for help with editing and further discussion. Work of the author was supported by NSF grants DMS-0943832 and DMS-1101373.

References

[1] H. Altman, Internal structure of addition chains: well-ordering, *Theoret. Comput. Sci.* (2017), `doi:10.1016/j.tcs.2017.12.002`

[2] H. Altman, Integer complexity and well-ordering, *Michigan Mathematical Journal* **64** (2015), no. 3, 509–538.

[3] H. Altman, Integer complexity: representing numbers of bounded defect, *Theoret. Comput. Sci.* **652** (2016), 64–85.

[4] H. Altman, Integer complexity: the integer defect, in preparation.

[5] H. Altman, Refined estimates for counting numbers of low defect, in preparation.

[6] H. Altman and J. Arias de Reyna, Integer complexity, stability, and self-similarity, in preparation.

[7] H. Altman and J. Zelinsky, Numbers with integer complexity close to the lower bound, *Integers* **12** (2012), no. 6, 1093–1125.

[8] J. Arias de Reyna, Complejidad de los números naturales, *Gac. R. Soc. Mat. Esp.* **3** (2000), 230–250.

[9] J. Arias de Reyna and J. Van de Lune, Algorithms for determining integer complexity, `arXiv:1404.2183`, 2014

[10] A. Brauer, On addition chains, *Bull. Amer. Math. Soc.*, **45** (1939), 736–739.

[11] P. W. Carruth, Arithmetic of ordinals with applications to the theory of ordered abelian groups, *Bull. Amer. Math. Soc.* **48** (1942), 262–271.

[12] J. H. Conway, *On Numbers and Games*, Second Edition, A K Peters, Ltd., Natick, Massachusetts, 2001, pp. 3–14.

[13] D. H. J. De Jongh and R. Parikh, Well-partial orderings and hierarchies, *Indag. Math.* **39** (1977), 195–206.

[14] H. Dellac, *Interméd. Math.* **1** (1894), 162–164.

[15] A. Flammenkamp, Drei Beiträge zur diskreten Mathematik: Additionsketten, No-Three-in-Line-Problem, Sociable Numbers, Diplomarbeit in Mathematics (Bielefield University, 1991), pp. 3–118.

[16] R. K. Guy, Some suspiciously simple sequences, *Amer. Math. Monthly*, **93** (1986), 186–190; and see **94** (1987), 965 & **96** (1989), 905.

[17] R. K. Guy, *Unsolved Problems in Number Theory*, Third Edition, Springer-Verlag, New York, 2004, pp. 399–400.

[18] J. Iraids, personal communication.

[19] J. Iraids, K. Balodis, J. Čerņenoks, M. Opmanis, R. Opmanis, K. Podnieks. Integer complexity: experimental and analytical results, `arXiv:1203.6462`, 2012

[20] D. E. Knuth, *The Art of Computer Programming*, Vol. 2, Third Edition, Addison-Wesley, Reading, Massachusetts, pp. 461–485

[21] K. Mahler and J. Popken, On a maximum problem in arithmetic (Dutch), *Nieuw Arch. Wiskunde*, (3) **1** (1953), 1–15; *MR* **14**, 852e.

[22] A. Scholz, Aufgabe 253, Jahresbericht der Deutschen Mathematikervereinigung, Vol. 47, Teil II, B. G. Teubner, Leipzig and Berlin, 1937, pp. 41–42.

[23] V. V. Srinivas & B. R. Shankar, Integer complexity: breaking the $\Theta(n^2)$ barrier, *World Academy of Science*, **41** (2008), 690–691

[24] M. V. Subbarao, Addition chains – some results and problems, *Number Theory and Applications*, Editor R. A. Mollin, NATO Advanced Science Series: Series C, V. 265, Kluwer Academic Publisher Group, 1989, pp. 555–574.

[25] I. Volkovich, Characterizing arithmetic read-once Formulae, *ACM Trans. Comput. Theory* **8** (2016), no. 1, Art. 2, 19 pp.

[26] J. Zelinsky, An upper Bound on integer complexity, in preparation

Appendix: Implementation Notes

In this appendix we make some notes about the attached implementation of the above algorithms and on other ways they could be implemented.

We have actually not implemented Algorithm 8 and Algorithm 9 in full generality, where L may be arbitrary; we have only implemented the case where $L=\infty$, the case where $L=\|n\|$ (computed beforehand), and the case of Algorithm 10.

As was mentioned in Section 2.3, the step size in the attached implementation has been fixed at $\alpha=\delta(2)$, with the sets B_α and T_α precomputed. Other integral multiples of $\delta(2)$ were tried, up to $9\delta(2)$ (since $10\delta(2)>1$ and thus is not a valid step size), but these all seemed to be slower, contrary to the author's expectation.

Another variation with a similar flavor is that one could write a version of these algorithms with nonstrict inequalities, computing numbers n with $\delta(n)\le r$ for a given r, rather than $\delta(n)<r$; see Appendix A of [3]. We make the following definition to formalize this.

Definition 19. For a real number $r\ge 0$, the set $\overline{A}_r$ is the set $\{n\in\mathbb{N}:\delta(n)\le r\}$. The set $\overline{B}_r$ is the set of all elements of $\overline{A}_r$ which are leaders.

Definition 20. A finite set $\mathcal{S}$ of low-defect pairs will be called a *covering set* for $\overline{B}_r$ if, for every $n\in\overline{B}_r$, there is some low-defect pair in $\mathcal{S}$ that efficiently 3-represents it. We will say $\mathcal{S}$ is a *good covering* of $\overline{B}_r$ if, in addition, every $(f,C)\in\mathcal{S}$ satisfies $\delta(f,C)\le r$.

Then, as per Appendix A of [3], good coverings of $\overline{B}_r$ exist, and only slight variations on the algorithms above are needed to compute them. However, this was not tried in this implementation.

It is also worth noting that the check for whether a given polynomial f 3-represents a given number n can also be sped up. If f is a low-defect polynomial with leading coefficent a, maximum coefficient A, and N terms, then

$$a3^{k_1+\ldots+k_r} \le f(3^{k_1}, \ldots, 3^{k_r}) \le NA3^{k_1+\ldots+k_r},$$

so we only need to search $(k_1, \ldots, k_r)$ with

$$\lceil \log_3 \frac{n}{NA} \rceil \le k_1 + \ldots + k_r \le \lfloor \log_3 \frac{n}{a} \rfloor,$$

a stricter condition than was described in the algorithms above. This improvement is, in fact, used in the attached implementation. It is also possible that there is a better way than brute force.

As was mentioned in Section 6, when running Algorithm 8 or Algorithm 9 with L finite, one can omit the 3-representation check at intermediate steps. We have only implemented this variant for Algorithm 10.

It was mentioned in Section 3.2 that considering "low-defect expression pairs" (E, C) or "low-defect tree pairs" (T, C) (where E is a low-defect expression, T is a low-defect tree, and $C \ge \|E\|$ or $C \ge \|T\|$, as appropriate) may be useful. In fact, the attached implementation works with a tree representation essentially the same as low-defect trees and low-defect tree pairs. Among other things, this makes it easy to find the minimal variables to be substituted into. If one were actually representing low-defect polynomials as polynomials, this would take some work. There is a slight difference in that, rather than simply storing a base complexity $C \ge \|T\|$, it stores for each vertex or edge – say with label $\|n\|$ – a number k such that $k \ge \|n\|$, unless we are talking about a non-leaf vertex and $n = 1$, in which case $k = 0$. We can then determine a C by adding up the values of k. That is to say, the complexity, rather than being attributed to the whole tree, is distributed among the parts of the tree responsible for it; this makes it easier to check for and remove redundant low-defect pairs.

It was also mentioned in Section 3.2 that one could use a representation similar to low-defect expressions, but with all the integer constants replaced with $+, \cdot, 1$-expressions for same. For example, instead of $2(2x+1)$, one might have $(1+1)((1+1)x+1)$. We have not implemented this, but doing this would have one concrete benefit – it would allow the algorithms above to not only determine the complexity of a given number n, but also to give a shortest representation (and analogously with stable complexity). The current implementation cannot consistently do this in a useful manner. For instance, suppose that we ran Algorithm 10 and found some k with $\|2^k\| = 2k - 1$. We might then look at the actual low-defect pair (f, C) that

3-represented it, to learn what this representation with only $2k-1$ ones is. But it might turn out, on inspection, that f was simply the constant 2^k; this would not be very enlightening. Using $+, \cdot, 1$-expressions would remedy this, as would having low-defect pairs keep track of their "history" somehow.

It is also possible to write numerical versions of Proposition 6, that say exactly how far out one has to go in order to get within a specified ε of the limit $\delta(f, C)$; one could use this in Algorithm 4 instead of simply searching larger and larger K until one works. This was tried but found to be slower.

Finally, it is worth expanding here on the remark in Section 2.4 that it is possible to write Algorithm 8 and Algorithm 9 without using truncation. Surprisingly little modification is required; the only extra step needed is that, in order to check if n (or any $3^k n$) has defect less than $i\alpha$, instead of just checking if a low-defect pair (f, C) (or its augmented version) 3-represents n (or any $3^k n$), if one finds that indeed $n = f(3^{k_1}, \ldots, 3^{k_r})$ (or the appropriate equivalent), one must additionally check whether $\delta_{f,C}(k_1, \ldots, k_r) < i\alpha$, since this is no longer guaranteed in advance. We will not state a proof of correctness here; it is similar to the proofs above. Such no-truncation versions of the algorithms were tried, but found to be too slow to be practical, because of the time needed to check whether the resulting polynomials 3-represented a given number. Another possibility, in the case where one is using a cutoff, is to truncate only at the final step, and not at the intermediate steps; this has not been tried. If this is used, it should probably be combined with not checking whether n (or any $3^k n$) is 3-represented until the final step, for the reason just stated.