ON ALGORITHMS TO CALCULATE INTEGER COMPLEXITY

KATHERINE CORDWELL, ALYSSA EPSTEIN, ANAND HEMMADY, STEVEN J. MILLER, EYVINDUR A. PALSSON, AADITYA SHARMA, STEFAN STEINERBERGER, AND YEN NHI TRUONG VU

ABSTRACT. We consider a problem first proposed by Mahler and Popken in 1953 and later developed by Coppersmith, Erdős, Guy, Isbell, Selfridge, and others. Let $f(n)$ be the complexity of $n \in \mathbb{Z}^+$, where $f(n)$ is defined as the least number of 1's needed to represent n in conjunction with an arbitrary number of $+$'s, $*$'s, and parentheses. Several algorithms have been developed to calculate the complexity of all integers up to n. Currently, the fastest known algorithm runs in time $\mathcal{O}(n^{1.230175})$ and was given by J. Arias de Reyna and J. Van de Lune in 2014. This algorithm makes use of a recursive definition given by Guy and iterates through products, $f(d)+f\left(\frac{n}{d}\right)$, for $d \mid n$, and sums, $f(a)+f(n-a)$ for a up to some function of n. The rate-limiting factor is iterating through the sums. We discuss improvements to this algorithm by reducing the number of summands that must be calculated for almost all n. We also develop code to run J. Arias de Reyna and J. van de Lune's analysis in higher bases and thus reduce their runtime of $\mathcal{O}(n^{1.230175})$ to $\mathcal{O}(n^{1.222911236})$.

1. INTRODUCTION

1.1. **Introduction.** In this paper, $\log$ denotes $\ln$, and $\log_b$ denotes the logarithm in base b. Given $n \in \mathbb{N}$, the complexity of n, which we denote $f(n)$, is defined as the least number of 1's needed to represent n using an arbitrary number of additions, multiplications, and parentheses. For example, because 6 may be represented as $(1+1)(1+1+1)$, $f(6) \leq 5$. Calculating $f(n)$ for arbitrary n is a problem that was posed in 1953 by Mahler and Popken [MP]. Guy [G] drew attention to this problem in 1986 when he discussed it and several other simply stated problems in an Am. Math. Monthly article. The following recursive expression for integer complexity highlights the interplay of additive and multiplicative structures:

$$f(n) = \min_{\substack{d \mid n \\ 2 \leq d \leq \sqrt{n} \\ 1 \leq a \leq n/2}} \left\{ f(d) + f\left(\frac{n}{d}\right), f(a) + f(n-a) \right\}. \tag{1.1}$$

Some unconditional bounds on $f(n)$ are known. In particular, [G] attributes a lower bound of $f(n) \geq 3\log_3(n)$ to Selfridge. Also, an upper bound of $f(n) \leq 3\log_2(n)$ is attributed to Coppersmith. Extensive numerical investigation (see [IBCOOP]) suggests that $f(n) \sim 3.3\log_3(n)$ for n large but it is not even known whether $f(n) \geq (3+\varepsilon_0)\log_3 n$ for some $\varepsilon_0 > 0$. As a step towards understanding these problems Altman and Zelinsky [AZ] introduced the discrepancy $\delta(n) = f(n) - 3\log_3(n)$ and provided a way to classify those numbers with a small discrepancy. This classification was taken further by Altman [A1, A2] where he obtained a finite set of polynomials that represent precisely the numbers with small defects. As a consequence Altman [A3] was able to calculate the integer complexity of certain classes of numbers. Any progress on these difficult questions likely requires a substantial new idea; the main difficulty, the interplay between additive and multiplicative structures, is at the core of a variety of different open problems, which we believe adds to its allure.

1.2. **Algorithms.** Much of the progress on this problem has been algorithmic. Using the above recursive definition, it is possible to write algorithms to calculate $f(n)$ for large values of n where the rate-limiting step of the algorithm is iterating through the summands, $f(a) + f(n-a)$, for many values of a. In particular, the brute-force algorithm that iterates over all $a's$ such that $1 \leq a \leq n/2$ runs in time $\mathcal{O}(n^2)$, but there are ways to bound the number of summands that must be checked so as to significantly decrease the computational complexity. Srinivar and Shankar [SS] used the unconditional upper and lower bounds on

Date: June 30, 2017.
2010 *Mathematics Subject Classification.* 11Y55, 11Y16 (primary) 11B75, 11A67, 68Q25 (secondary).
Key words and phrases. Integer Complexity.

$f(n)$ to bound the number of summands, obtaining an algorithm that runs in time $\mathcal{O}(n^{\log_2(3)}) < \mathcal{O}(n^{1.59})$.

The fastest known algorithm runs in time $\mathcal{O}(n^{1.230175})$ and is due to J. Arias de Reyna and J. van de Lune [AV]. Also, the experimental data in [IBCOOP] is based on an algorithm that calculates $f(n)$ for n up to $n = 10^{12}$. They derive many interesting results from their data, but they do not analyze the runtime of their algorithm. We obtain both an overall improvement on the runtime of the J. Arias de Reyna and J. van de Lune algorithm and an internal improvement to the workings of the algorithm. The overall improvement is derived from running the analysis of [AV] in much higher bases, while the internal improvement reduces the number of summands $f(a) + f(n-a)$ that must be calculated for almost all n. We detail these improvements in Sections 2 and 3, test the internal improvement in section 4, and end the paper by proposing a new approach for improving the current unconditional upper bound on $f(n)$.

2. Algorithmic aspects

2.1. **The de Reyna & van de Lune algorithm.** J. Arias de Reyna and J. van de Lune [AV] developed code in Python to perform the analysis of their algorithm, which they have generously shared with us. Additionally, Fuller has published open-source code [F] written in C to calculate integer complexities. Using these, we have developed code in C (see Appendix A) that is comparable to J. Arias de Reyna and J. van de Lune's Python code. The heart of the code is the calc_count method, which calculates $D(b,r)$ for varying values of b and r, where $D(b,r)$ is the smallest integer satisfying

$$f(r+bn) \leq f(n) + D(b,r) \tag{2.1}$$

for all n. These integers $D(b,r)$ are useful for bounding $f(n)$ in the following way: [AV] defined C_{avg} as the infimum of all C such that $f(n) \leq C\log(n)$ for a set of natural numbers of density 1 and showed that

$$C_{avg} \leq \frac{1}{b\log(b)} \sum_{r=0}^{b-1} D(b,r). \tag{2.2}$$

In this calculation, we refer to b as the base in which we are working. Our code closely follows the logic of J. Arias de Reyna and J. van de Lune's program, making the following slight optimization.

Theorem 2.1. *Take $b = 2^i3^j$ where $b < 10^{12}$ and $i + j > 0$. If $\mod(b,r) = 0$ for $2 \leq r < b$, then $D(b,r) = f(b) + 1$.*

Proof. Write $r(1+\frac{b}{r})$. From [IBCOOP], we know that $f(2^a3^b) = 2a+3b$ for $2^a3^b < 10^{12}$ and $a+b > 0$. We know that $b = 2^i3^j$, so $r \mid b$ is of the form 2^x3^y for $x \leq i$, $y \leq j$. This means that b/r is of the form $2^{i-x}3^{j-y}$. Since $r > 0$, $x + y > 0$ and since $r < b$, $(i-x) + (j-y) > 0$.

Then we obtain

$$1 + f(r) + f\left(\frac{b}{r}\right) = 1 + 2x + 3y + 2(i-x) + 3(i-y) = 1 + 2i + 3j = 1 + f(b). \tag{2.3}$$

This shows that $D(b,r) \leq f(b) + 1$. It is evident that $D(b,r) > f(b)$, because if we multiply $n \in \mathbb{N}$ by b, then we have to use at least $f(b)$ 1's. If we multiply by b and also add r where $r > 1$, then we have to use at least one additional 1 for the addition step (because at some point we need something of the form $k(x + y(z))$, where $kx = r$, $ky = b$, and z is any number). $\square$

J. Arias de Reyna and J. van de Lune [AV] suggest that their algorithms will be more powerful when implemented in C and Pascal. [AV] calculated the runtime of their algorithm for bases 2^n3^m up to 3188246, and found the best value $\mathcal{O}(n^{1.230175})$ in base $2^{10}3^7 = 2239488$. Using C is advantageous because it runs much faster than Python, and so we are able to calculate values for higher bases. We calculate values for bases $2^n3^m \leq 57395628$. In base $2^{13}3^8 = 53747712$, we find that the runtime is $\mathcal{O}(n^{1.222911236})$.

2.2. **Optimal asymptotic results.** Guy [G] was the first to remark that while pointwise bounds seem difficult, it is possible to establish bounds that are true for almost all (in the sense of asymptotic density 1) numbers. His method showed that $f(n) \leq 3.816\log_3 n$ for a subset of integers with density 1. As a consequence of out computations, it follows that we can improve this result to $f(n) \leq 3.61989\log_3(n)$ by performing the computations in base $2^{11}3^9$.

3. IMPROVED RESULTS VIA BALANCING DIGITS

3.1. **Balancing Digits.** We improve the algorithm for calculating complexity given in [AV]. The rate-limiting factor in this algorithm is checking, for all $n \leq N$, $f(a) + f(n-a)$ for all $1 \leq a \leq k_{\text{Max}}$ for some k_{Max} that is a function of n. We will show that we can reduce the number of summands that must be checked for almost all n. We say that $n \in \mathbb{Z}$ is *digit-balanced* in base b if each of the digits $1, \dots, b-1$ occurs roughly $1/b$ times in the base b representation of n, or *digit-unbalanced* if some digits occur significantly more often than others. We will show that almost all numbers are digit-balanced, although the exact threshold of variation that we allow will depend on the base b. Finally, assuming that we have a set S of digit-balanced numbers in base b, we will use Guy's method to find that for any $n \in S$, $f(n) \leq c \log_3(n)$ for some c. Then, using this bound on $f(n)$ and assuming that $f(n) = f(a) + f(n-a)$, we are able to bound a, which, in turn, narrows the search space that a reasonable algorithm has to cover.

3.2. **Bounds on Digit-Balanced Numbers.** Our main result is as follows.

Proposition 3.1. *There exists a constant $c_b > 0$ only depending on the base b such that*

$$\#\left\{1 \leq n \leq N : \max_{1 \leq i \leq b} \left| \frac{\textit{number of digits of } n \textit{ in base } b \textit{ that are } i}{\textit{number of digits of } n \textit{ in base } b} - \frac{1}{b} \right| \geq \varepsilon \right\} \leq N^{1 - c_b \varepsilon^2}.$$

Proof. The main idea behind the argument is to replace a combinatorial counting argument by the probabilistic large deviation theory. Let $N = b^k$, and consider all k-digit numbers in base b, let X_i be a random variable such that $X_i = 1$ with probability $1/b$ and 0 otherwise for $1 \leq i \leq k$. For any given digit $0 \leq d < b$, each X_i gives the probability that this digit will appear in a fixed position i in the base b representation of a number. Since we are considering k-digit numbers, we need to understand the average value of $X_1 + \cdots + X_k$ and to analyze how close this average is to $\frac{1}{b}$. Let $\overline{X} = \frac{1}{k}(X_1 + \cdots + X_k)$. Next, we can use Hoeffding's inequality, which gives

$$P(\overline{X} - \frac{1}{b} \geq \epsilon) \leq e^{-2k\epsilon^2}. \tag{3.1}$$

We know that $k \approx \log_b(N) = \frac{\log(N)}{\log(b)}$, so:

$$e^{-2k\epsilon^2} = e^{-2\epsilon^2 \frac{\log(N)}{\log(b)}} = \left(e^{\log(N)}\right)^{-2\epsilon^2 \frac{1}{\log(b)}} = N^{\frac{-2\epsilon^2}{\log(b)}}. \tag{3.2}$$

So, the probability that a number with k digits in its base b representation has some digits that appear more often than average is less than or equal to $N^{\frac{-2\epsilon^2}{\log(b)}}$, meaning that $|S| \leq N \cdot N^{\frac{-2\epsilon^2}{\log(b)}} = N^{1-\frac{2\epsilon^2}{\log(b)}}$. $\square$

3.3. **Bound on Number of Summands.** Assume now that $f(n) = f(n-a) + f(a)$ and that this is the optimal representation using the least number of 1's. We abbreviate $f(n) = c\log_3(n)$and want to derive a bound on a. The main idea is to show that the logarithmic growth implies that a cannot be very large (otherwise the growth of $f(n)$ would be closer to linear). Using the lower bound due to Selfridge [G], we attain:

$$c\log_3(n) \geq 3(\log_3(n-a) + \log_3(a)). \tag{3.3}$$

This is equivalent to:

$$\log_3(n^{c/3}) \geq \log_3(n-a) + \log_3(a). \tag{3.4}$$

Say that $a = qn$, where necessarily $q \leq \frac{1}{2}$. Then we have:

$$\log_3(n^{c/3}) \geq \log_3((1-q)n \cdot a). \tag{3.5}$$

Exponentiating both sides and simplifying gives

$$\frac{n^{c/3-1}}{1-q} \geq a. \tag{3.6}$$

Since $q \leq \frac{1}{2}$, then $1 - q \geq \frac{1}{2}$, and so

$$\frac{n^{c/3-1}}{1/2} \;\geq\; \frac{n^{c/3-1}}{1-q} \;\geq\; a, \tag{3.7}$$

or:

$$2n^{c/3-1} \;\geq\; \frac{n^{c/3-1}}{1-q} \;\geq\; a. \tag{3.8}$$

Thus, we need only check for values of a at most $2n^{c/3-1}$.

3.4. **Binary Analysis.** To see how our improvement works, we analyze it in the simplest possible base, which is binary. Consider k-digit numbers less than N (so that $k \;\approx\; \log_2(N)$). The average case in Guy's method, illustrated in [G] and based on Horner's scheme of representing binary numbers, gives $f(n) \leq 5\log_2(n)/2$, or $f(n) < 3.962407\log_3(n)$. "Bad" numbers in base 2 are those that have many 1's, as that is when the representation is rather inefficient. If we move away from the average case to numbers which have, say, 75% 1's and 25% 0's, then the constant in Guy's method is

$$\frac{1}{\ln(2)}(3\cdot .75 + 2\cdot .25)\ln(3) \;<\; 4.358647. \tag{3.9}$$

This is already much worse than the original average case constant of 3.962407, and so we need to stay much closer to the average case. In particular, the following percentages of 1's and 0's give the following values for the constant in Guy's method:

Percent 0's	Percent 1's	Constant
46	54	4.02581
47	53	4.00997
48	52	3.99411
49	51	3.97826
49.9	50.1	3.96399
49.99	50.01	3.962565

In fact, we have an improvement on the original algorithm on numbers with at most 46% 0's and 54% 1's, because the analysis from the previous section affords a bound of $a \leq 2n^{4.02581/3-1} \leq 2n^{0.342}$ for such numbers. Next we want to understand how often this case occurs. We need to bound the number of times that 0 occurs at most $\frac{46N}{100}$ times. So, letting $n-d \;=\; \frac{46N}{100}$, we need to bound

$$B(n,0)+\cdots+B(n,n-d), \tag{3.10}$$

where B denotes the binomial distribution. We can bound this by

$$B(n,0)+\cdots+B(n,n-d) \;\leq\; e^{-nD\left(\frac{n-d}{n}||\frac{1}{2}\right)} \tag{3.11}$$

where

$$D\left(\frac{n-d}{n}\;||\;\frac{1}{2}\right) \;=\; \frac{n-d}{n}\log\left(2\left(\frac{n-d}{n}\right)\right)+\left(1-\frac{n-d}{n}\right)\log\left(2\left(1-\frac{n-d}{n}\right)\right). \tag{3.12}$$

Because $n \;\approx\; \log_2(N)$, we get that

$$B(n,0)+\cdots+B(n,n-d) \;\leq\; N^{-D(\frac{n-d}{n}\;||\;\frac{1}{2})\frac{1}{\log 2}}. \tag{3.13}$$

In particular, then, there are at most $N^{1-D(\frac{n-d}{n}||\frac{1}{2})}$ "bad" numbers, where here $\frac{n-d}{n} \;=\; \frac{46}{100}$. Plugging this in yields $< N^{1-.003203}$ bad numbers, i.e. we have improved the other $> N^{.003203}$ numbers, which is significant as N grows large. This is the analysis for the binary case. The ideal bases are, of course, much larger than binary, which makes analysis complicated. Accordingly, to understand how our algorithm works in the general case, we performed a number of empirical tests.

4. Empirical Calculations

To see whether our method improves J. Arias de Reyna and J. van de Lune's algorithm in practice, we modified J. Arias de Reyna and J. van de Lune's code by adding various precomputations and calculating how many numbers would be improved with these precomputations.

The first precomputation uses a greedy algorithm due to Steinerberger [St], which gives that $f(n) \leq 3.66\log_3(n)$ for most n. The recursive algorithm works as follows: if $n \equiv 0 \mod 6$ or $n \equiv 3 \mod 6$, take $n = 3(n/3)$ and run the algorithm on $n/3$. If $n \equiv 2 \mod 6$ or $n \equiv 4 \mod 6$, take $n = 2(n/2)$ and run the algorithm on $n/2$. If $n \equiv 1 \mod 6$, take $n = 1 + 3(n-1)/3$ and run the algorithm on $(n-1)/3$. If $n \equiv 5 \mod 6$, take $n = 1 + 2(n-1)/2$ and run the algorithm on $(n-1)/2$.

The method is as follows: First, run the greedy algorithm on all of the numbers up to some limit and store the results in a dictionary. Then, use these values to compute a bound on the number of summands for each number (using the formula derived in Section 3.3). Store a counter that is initialized to 0. Next, run J. Arias de Reyna and J. van de Lune's algorithm. For each number, test whether the precomputed summand bound is better than the summand bound in the original algorithm. If an improvement is found, increment the counter. When we use this algorithm to precompute summands, we improve 7153 numbers out of the first 200000, or less than 3.6% of numbers. If we compute complexities further, up to 2000000, we improve 60864 numbers, or less than 3.05% of numbers. See Appendix B for the code.

We can also combine Steinerberger's algorithm with a stronger algorithm, due to Shriver [Sh]. Shriver developed a greedy algorithm in base 2310. If we use the best upper bound on complexities from Shriver and Steinerberger's greedy algorithm, we improve 11188 numbers out of 200000, or about 5.6% of numbers. If we compute complexities up to 2000000, we improve 107077 numbers, or less than 5.36% of numbers.

Shriver conjectures that his best algorithm, which uses simulated annealing, produces a bound of $f(n) \leq 3.529\log_3(n)$ for generic integers. In fact, only 824 numbers up to 2000000 would be improved by assuming a uniform bound of $f(n) \leq 3.529\log_3(n)$. Of course, this is a purely theoretical result—if we were to actually introduce a uniform bound, then we would not be able to accurately calculate complexities. If we become even more optimistic and use a uniform bound of $f(n) \leq 3.5\log_3(n)$, we would only potentially improve 4978 numbers out of the first 2000000. Similarly, using $f(n) \leq 3.4\log_3(n)$ would improve 124707 numbers of 2000000, which is about 6.23%. If we venture significantly below Shriver's conjecture of $3.529\log_3(n)$ and use $f(n) \leq 3.3\log_3(n)$ uniformly, then we start to see a significant difference—we would improve 726756 numbers of 2000000, or about 36%.

Overall, it seems that Arias de Reyna and van de Lune's algorithm already has a strong bound on the number of summands that are computed. It is possible that the complexity of J. Arias de Reyna and J. van de Lune's algorithm is significantly lower than $\mathcal{O}(n^{1.223})$. Thus, while summand precomputing improves the complexity computation for some numbers, given the overhead for performing precomputations and the current speed of J. Arias de Reyna and J. van de Lune's algorithm, introducing a precomputation does not seem to yield an overall improvement to the algorithm.

5. Progress Towards an Unconditional Upper Bound

The current unconditional upper bound on complexity, $f(n) \leq 3\log_2(x)$, is derived from applying Guy's method in base 2 to n. In particular, the most complex numbers have binary expansions of the form $11\cdots1_2$ so that at each step, Guy's method requires three 1's. The resulting representation is of the form $1 + (1+1)[1 + (1+1)[\cdots]]$.

Say that $n \bmod 3 \equiv k$. Instead of applying Guy's method to n, what if write $n = k + (1+1+1)(n-k)/3$ and then apply Guy's method to $(n-k)/3$? Then in the case where $n = 11\cdots1_2$, $(n-k)/3$ is either of the form $1010\cdots1$ or $1010\cdots0$, and applying Guy's method to $(n-k)/3$ gives $f((n-k)/3) \leq 1 + 2.5\log_2(n)$. Using this, we find that $f(n) \leq 6 + 2.5\log_2(n)$, which is a significant improvement over $f(n) \leq 3\log_2(n)$.

This suggests the following method: If the binary representation of n contains more than a certain percentage of 1's, then write n as $k + (1+1+1)\cdot(n-k)/3$ and apply Guy's method instead to $(n-k)/3$. Empirically, in most cases, when the binary expansion of n contains a high percentage of 1's, $(n-k)/3$ has a significantly lower percentage of 1's. However, there are some examples where this fails. For example, if $n = 2^{102} - 2^{100} - 2$, then both the binary expansion of n and the binary expansion of $(n-1)/3$ have a high percentage of 1's. Notably, if we repeat this division process and consider $((n-1)/3)/3$, then we will obtain a number with a nice binary expansion. Accordingly, we say that $2^{102}-2^{100}-2$ requires two iterations of division by 3.

Some numbers require numerous iterations of division by 3 before their binary expansions are nice. For example, $n=2^{3000}-2^{2975}-2^{2807}-1$ requires nine iterations. These sorts of counterexamples seem to follow some interesting patterns. Let n_i denote the number obtained after i iterations of division by 3 so that $n_0=n$, $n_1=(n_0-(n_0 \bmod 3))/3$, etc. In general, it seems that the number of iterations that are necessary to produce a "nice" binary expansion is tied to the number of iterations for which $n \equiv 2 \bmod 3$. For example, when $n=2^{3000}-2^{2975}-2^{2807}-1$, then $n_0 \equiv n_1 \equiv n_2 \equiv \cdots \equiv n_7 \equiv 2 \bmod 3$, but $n_8 \equiv 0 \bmod 3$, and n_9 has the first "nice" binary expansion.

It should be noted that there is no reason to only employ division by 3. For example, when $n=2^{3000}-2^{2975}-2^{2807}-1$, $n \bmod 11 \equiv 5$, and $(n-6)/11$ has a nice binary expansion. It should be noted that $n \equiv 4 \bmod 5$ and $n \equiv 6 \bmod 7$, and the binary representations of $(n-4)/5$ and $(n-6)/7$ both contain a large percentage of 1's.

In general, then, performing this process of division by appropriate numbers before applying Guy's method is a promising strategy for obtaining an improvement on the unconditional upper bound on $f(n)$. We believe that it could be an interesting problem to make these vague heuristics precise and understand whether this could give rise to a new effective method of giving explicit constructions of n with sums and products that use few $1's$.

6. Acknowledgments

We would like to thank Professor Arias de Reyna for generously sharing the code that he developed with Professor van de Lune. Thank you to the SMALL REU program, Williams College, and the Williams College Science Center. We would like to thank Professor Amanda Folsom for funding as well as NSF Grants DMS1265673, DMS1561945, DMS1347804, DMS1449679, the Williams College Finnerty Fund, and the Clare Boothe Luce Program. Finally the fifth listed author was supported in part by Simons Foundation Grant #360560.

APPENDIX A. IMPLEMENTATION OF THE ALGORITHM IN C

```
/* A significant portion of this code is due to Fuller; see
http://oeis.org/A005245/a005245.c.txt */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef unsigned char A005245_value_t;
typedef struct { unsigned size; A005245_value_t *array; } A005245_array_t;
A005245_array_t * A005245_arr;
int powarrlen = 286;
/* Store all powers 2^m 3^n where 2^m*3^n < 500000000 */
int powarr[286] = {2, 3, 4, 6, ..., 483729408};
#define MAX_A005245_VALUE 127
/* Keep a safety factor of 2 to avoid overflow */

unsigned A000792(A005245_value_t n)
{
  unsigned result = 1;
  while (n >= 5 || n == 3)
    {
      result *= 3;
      n -= 3;
    }
  return result << (n/2);
}

/* Helper method, specific to powarr */
int index_of(int val) {
  for (int i = 0; i < powarrlen; i++) {
    if (powarr[i] == val) {
      return i;
    }
  }
  /* if the element is not found */
  return -1;
}

void A005245_free(A005245_array_t *a)
{
  a->size = 0;
  free(a->array);
  a->array = NULL;
}

/* Thanks to http://stackoverflow.com/questions/5666214/
how-to-free-c-2d-array */
void COMPL_free(int **a)
{
  for (int i = 0; i < powarrlen; i++)
    {
      int* currRowPtr = a[i];
```

```
      free(currRowPtr);
    }
  free(a);
  a = NULL;
}

int A005245_init(A005245_array_t *a, unsigned size)
{
  unsigned i;

  /* This line was in Fuller's original code but seemed to cause problems */
  // A005245_free(a);
  a->array = (A005245_value_t*)malloc(size * sizeof(A005245_value_t));
  if (a->array)
    {
      a->size = size;
      a->array[1] = 1;
      for (i = 2; i < size; i++)
        a->array[i] = MAX_A005245_VALUE;
    }
  return (a->array != 0);
}

/* Thanks to https://www.eskimo.com/~scs/cclass/int/sx9b.html */
int ** COMPL_init()
{
  int ** a;
  int i, j, val_at_index;

  /* COMPL_free(a); */
  /* There are powarrlen rows */
  a = (int**) calloc(powarrlen, sizeof(int *));
  if (a == NULL) {
    fprintf(stderr, "out of memory\n");
    exit(1);
  }

  for (i = 0; i < powarrlen; i++) {
    val_at_index = powarr[i];
    a[i] = (int*) calloc(val_at_index, sizeof(int));
    if (a[i] == NULL) {
      fprintf(stderr, "out of memory\n");
      exit(1);
    }
  }

  /* initialize D(2, 0) and D(2, 1) specially */
  a[0][0] = 2;
  a[0][1] = 3;

  return a;

}
```

```
void A005245_additions_to_n(A005245_array_t *a, unsigned n)
{
  unsigned limit_m, m;
  A005245_value_t target, k;

  if (a->array[n] > a->array[n-1] + 1)
    a->array[n] = a->array[n-1] + 1;

  target = a->array[n-1];
  k = target / 2;
  while (A000792(k) + A000792(target - k) < n)
    k--;
  limit_m = A000792(k);
/* Already used m=1 earlier, and don't need m=2..5 as they cannot be better
than m=1 */
  for (m = 6; m <= limit_m; m++)
    {
      if (a->array[n] > a->array[m] + a->array[n-m])
        {
          printf("Counterexample to [3]: A(%u) + A(%u) = %u < conjecture(%u)
           = %u\n", m, n-m, (unsigned)(a->array[m] + a->array[n-m]), n,
                  (unsigned)a->array[n]);
          a->array[n] = a->array[m] + a->array[n-m];
        }
      else if (a->array[n-1]+1 > a->array[m] + a->array[n-m])
        printf("Counterexample to [2]: A(%u) + A(%u) = %u < A(%u-1)+1 = %u\n",
               m, n-m, (unsigned)(a->array[m] + a->array[n-m]), n,
               (unsigned)a->array[n-1]+1);
    }
}

void A005245_multiplications_from_n(A005245_array_t *a, unsigned n)
{
  unsigned m, mn;
  for (m = 2, mn = 2*n; (m <= n) && (mn < a->size); m++, mn += n)
    if (a->array[mn] > a->array[m] + a->array[n])
      a->array[mn] = a->array[m] + a->array[n];
}

/* Thanks to http://stackoverflow.com/questions/213042/
how-do-you-do-exponentiation-in-c */
int power(int base, unsigned int exp) {
  int i, result = 1;
  for (i = 0; i < exp; i++)
    result *= base;
  return result;
}

/* Thanks to http://stackoverflow.com/questions/5281779/
c-how-to-test-easily-if-it-is-prime-number */
int is_prime(int num)
{
  if (num <= 1) return 0;
  if (num % 2 == 0) return 0;
```

```
  for(int i = 3; i <= num/2; i+= 2)
    {
      if (num % i == 0)
        return 0;
    }
  return 1;
}

/* This function should update count */
int calc_count(int b, int r, int calc, A005245_array_t *a, int ** COMPL_arr) {
  int i, j, calc1, calc2, calc3, pow, base_index;

  base_index = index_of(b);
  /* Update calc with complexity of b + complexity of r, if appropriate */
  if (is_prime(b) == 1) {
    calc = a->array[b] + a->array[r];
  }
  else {
    if (r == 0) {
      calc = a->array[b];
    }
    else if (r == 1) {
      calc = a->array[b] + 1;
    }
    else if ((b % r) == 0) {
      calc = a->array[b] + 1;
    }
    /* Recursive step--access COMPL_arr */
    else if (COMPL_arr[base_index][r] != 0) {
      calc = COMPL_arr[base_index][r];
    }
    else {
      if (a->array[b] + a->array[r] < calc) {
        calc = a->array[b] + a->array[r];
      }

      /* We are trying to run up to a large base so we check under large
      powers of 2, 3 */
      for (i = 0; i < powarrlen; i++) {
        pow = powarr[i];
        if (pow < b) {
          if ((b % pow == 0) && (pow > 1)) {
            calc1 = calc_count(pow, r%pow, calc, a, COMPL_arr);
            calc2 = calc_count(b/pow, floor(r/pow), calc, a, COMPL_arr);
            /* Add the two together */
            calc3 = calc1 + calc2;
            /* Test this against the original calc */
            if (calc > calc3) {
              calc = calc3;
            }
          }
        }
      }
    }
```

```
  }

  return calc;
}

int main(int argc, char *argv[]) {
  int ret, r, i, j, arr_base, n = 483729409;
  unsigned long int p = 0, m0 = 0, m1 = 0, m2 = 0;
  A005245_array_t A005245;
  int ** COMPL_arr = COMPL_init();
  FILE *f = fopen("fuller_beyond.txt", "w");

  if (!A005245_init(&A005245, n)) {
      A005245_free(&A005245);
      printf("Not enough memory\n");
      return 2;
    }

  for (n = 2; n < A005245.size; n++)
    {
      A005245_additions_to_n(&A005245, n);
      A005245_multiplications_from_n(&A005245, n);
    }

  /* We have already initialized powarr[2, 0] and powarr[2, 1] */
  for(i = 1; i < powarrlen; i++) {
    arr_base = powarr[i];
    for (j = 0; j < arr_base; j++) {
      r = calc_count(arr_base, j, 1000000, &A005245, COMPL_arr);
      COMPL_arr[i][j] = r;
      if (4294967294 - p < r) {
        printf("Unsigned int limit is about to overflow; output past this
         point is unreliable");
      }
      p += r;
      if (r%3 == 0) {
        m0 += pow(3, (r/3));
      }
      if (r%3 == 1) {
        m1 += pow(3, ((r-1)/3));
      }
      if (r%3 == 2) {
        m2 += pow(3, ((r-2)/3));
      }
    }
    /* Write output to the terminal */
    printf("p is: %lu, base is: %d\n", p, arr_base);
    printf("Base is: %d, %lu, %lu, %lu\n", arr_base, m0, m1, m2);
    /* Write output to a file */
    fprintf(f, "p is: %lu, base is: %d\n", p, arr_base);
    fprintf(f, "Base is: %d, %lu, %lu, %lu\n", arr_base, m0, m1, m2);
    fflush(f);
    p = 0;
    m0 = 0;
```

```
    m1 = 0;
    m2 = 0;
    fflush(stdout);
  }

  fclose(f);
  A005245_free(&A005245);
  COMPL_free(COMPL_arr);

  return 0;
}
```

APPENDIX B. EMPIRICALLY TESTING SUMMAND PRECOMPUTATION

```
/* Much of this code is due to Arias de Reyna and van de Lune.  In particular,
their code has been modified to include a precomputation in order to test how
often a better summand bound is obtained.  This particular piece of code uses
Steinerberger's Markov chain algorithm in the precomputation.*/

from __future__ import division
import sys
sys.path.insert(0,'/Volumes/RIEMANN/Todo/Matematicas/Python')
from A000792 import *
from fractions import Fraction
from math import ceil
from math import log
from math import floor
from math import *

markov_dict = {}

def markovbound(n):
    count = 0
    if n <= 5:
        count = n
    else:
        if n%6 == 0 or n%6 == 3:
            count = 3 + markov_dict[n/3]
        if n%6 == 1:
            count = 4 + markov_dict[(n-1)/3]
        if n%6 == 2 or n%6 == 4:
            count = 2 + markov_dict[n/2]
        if n%6 == 5:
            count = 3 + markov_dict[(n-1)/2]
    markov_dict[n] = count

for i in range (1, 2000001):
    markovbound(i)

def precompute_summands(n):
    c = markov_dict[n]
    # need to make sure it never rounds down
    log_bound = c/float(log(n, 3)) + 1/float(100000000000)
    bound = float(log_bound)/3 - 1
```

```
    return bound

def compute_complexities(num):
    # count how often the new summand bound kicks in
    numImproved = 0
    nMax = num
    cMax = int(ceil(3.*log(nMax)/log(2)))
    print 'cMax = ', cMax

    # initialize the dictionary
    Compl={}
    for n in range(2,nMax+1):
        Compl[n]=cMax
    Compl[1]=1
    # now Compl[n] is always greater or equal to the true complexity
    for n in range(2,nMax+1):
        # test the sums
        a = Compl[n-1]+1
        if a < Compl[n]:
            Compl[n]=a  # the usual best value
        target = Compl[n-1]
        k=target/2
        while (A000792(k)+A000792(target-k)<n):
            k=k-1
        limitm = A000792(k)

        # do the summand precomputation
        markov_sumbound = precompute_summands(n)
        up_to = int(ceil(n**markov_sumbound))

        # check how many numbers are improved
        if (up_to < limitm):
            numImproved += 1

        # now test for sums
        limitm = min(limitm, up_to)
        for m in range(6,limitm+1):
            sumvalue = Compl[m]+Compl[n-m]
            if sumvalue < Compl[n]:
                Compl[n] = sumvalue
        # test for the products
        for k in range(2, min(n,nMax/n)+1):
            prodvalue = Compl[k]+Compl[n]
            if prodvalue <Compl[k*n]:
                Compl[k*n]=prodvalue
    # end of computation of complexities
    print "Computed complexities upto nMax Compl[",nMax,"] = ",  Compl[nMax]
    print "Num improved ", numImproved

# run the program
compute_complexities(2000000)
```

REFERENCES

[A1] H. Altman, *Integer complexity and well-ordering*, Michigan Math. Journal **64** (2015), no. 3, 509-538.

[A2] H. Altman, *Integer complexity: Representing numbers of bounded defect*, Theoretical Computer Science **652** (2016), 64-85.

[A3] H. Altman, *Integer complexity: Algorithms and computational results*, 2016. arXiv:1606.03635.

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

[AV] J. Arias de Reyna & J. Van de Lune, *Algorithms for Determining Integer Complexity*, arXiv:1404.2183 [math.NT].

[F] M. N. Fuller, *C-Program to Compute A005245*, Feburary 2008. http://oeis.org/A005245/a005245.c.txt.

[G] R. K. Guy, *Unsolved problems: Some suspiciously simple sequences*, Amer. Math. Monthly **93** (1986), no. 3, 186-190.

[IBCOOP] J. Iraids, K. Balodis, J.Cerenoks, M. Opmanis, R. Opmanis, and K. Podnieks, *Integer complexity: Experimental and analytical results*, Scientific Papers University of Latvia,Computer Science and Information Technologies **787** (2012), 153-179.

[MP] K. Mahler and J. Popken, *On a maximum problem in arithmetic. (Dutch)*, Nieuw Arch. Wiskd. **3** (1953), no. 1, 1-15.

[Sh] C. Shriver, *Applications of Markov Chain Analysis to Integer Complexity*, 2016. arXiv:1511.07842.

[St] S. Steinerberger, *A short note on integer complexity*, Contributions to Discrete mathematics, **9** (2014), no. 1.

[SS] V. V. Srinivas and B. R. Shankar, *Integer complexity: Breaking the $\theta(n^2)$ barrier*, World Academy of Science, Engineering and Technology **2** (2008), no. 5, 454 - 455.

E-mail address: ktcordwell@gmail.com

DEPARTMENT OF MATHEMATICS, UNIVERSITY OF MARYLAND, COLLEGE PARK, 20742

E-mail address: ale2@williams.edu

DEPARTMENT OF MATHEMATICS AND STATISTICS, WILLIAMS COLLEGE, WILLIAMSTOWN, MA 01267

E-mail address: ash6@williams.edu

DEPARTMENT OF MATHEMATICS AND STATISTICS, WILLIAMS COLLEGE, WILLIAMSTOWN, MA 01267

E-mail address: sjm1@williams.edu, Steven.Miller.MC.96@aya.yale.edu

DEPARTMENT OF MATHEMATICS AND STATISTICS, WILLIAMS COLLEGE, WILLIAMSTOWN, MA 01267

E-mail address: palsson@vt.edu

DEPARTMENT OF MATHEMATICS, VIRGINIA TECH, BLACKSBURG, VA 24061

E-mail address: as17@williams.edu

DEPARTMENT OF MATHEMATICS AND STATISTICS, WILLIAMS COLLEGE, WILLIAMSTOWN, MA 01267

E-mail address: stefan.steinerberger@yale.edu

DEPARTMENT OF MATHEMATICS, YALE UNIVERSITY, NEW HAVEN, CT 06510

E-mail address: ytruongvu17@amherts.edu

DEPARTMENT OF MATHEMATICS, AMHERST COLLEGE, AMHERST, MA 01002