Journal of Integer Sequences, Vol. 26 (2023),
Article 23.5.4

The Largest Integer Not the Sum of Distinct 8th Powers

Michael J. Wiener
20 Hennepin St.
Nepean, ON K2J 3Z4
Canada
michael.james.wiener@gmail.com

Abstract

The largest integer that is not the sum of one or more distinct squares is 128 and is called the threshold of completeness for the set of squares. A natural extension is to consider positive cubes and higher powers. Previous research solved this problem for powers up to 7. Our contribution is the threshold of completeness for 8th powers as well as lower bounds for powers from 9 to 16. We also describe the mathematical methods we used to speed up computations. Using 200,000 computer cores, our methods could find the threshold of completeness for 9th powers in about a month, and for 10th powers in about a year.

1 Introduction

There is no way to write 128 as the sum of one or more distinct squares, and it is the largest such number [14]. The fact that the summed squares are distinct is important; $128 = 8^2 + 8^2$ does not count. We say that 128 is not *summable* with the set $\{1^2, 2^2, \ldots\}$, and is the largest such number. We can prove this by first showing that 128 is not summable with squares and then showing that each integer from 129 to 249 is summable with the finite set $\{1^2, 2^2, \ldots, 10^2\}$. We can then add 11^2 to each of the solutions for the integers from 129 to 249 to cover the range 250 to 370. We then add 12^2 to the solutions for the integers from 129 to 370 to extend the covered range. Continuing with $13^2, 14^2, \ldots$, the covered range extends indefinitely with overlaps but without leaving any gaps. We call 128 the *threshold of completeness* for squares.

In this work, we focus on the threshold of completeness for sets of positive nth powers. However, we can generalize the idea of thresholds of completeness to other sets, such as the triangular numbers, $(x(x+1)/2)_{x\in\mathbb{N}} = \{1, 3, 6, 10, \ldots\}$, or to any other polynomial. Fuller and Nichols studied thresholds of completeness for a wide range of generalizations of this problem including sets that do not include all powers, and requiring that the minimum number of powers summed be some constant other than 1 [2]. They use the term *anti-Waring number* for the smallest number such that it and every subsequent integer is summable. Thus the anti-Waring number is one more than the threshold of completeness (when both numbers exist).

The threshold of completeness does not exist for all sets because there might be no point beyond which all integers are summable. For example, the set of even numbers cannot give any odd sums, and any finite set only gives a finite number of sums. However, Sprague proved that the threshold of completeness for positive nth powers exists for all positive integers n [15], and Roth and Szekeres generalized this proof by showing which polynomials $p(x)$ are such that $(p(x))_{x\in\mathbb{N}}$ has a threshold of completeness [11]. Later, Graham gave an alternative proof of Roth and Szekeres' result in an "elementary manner" [3].

Kim made an interesting contribution proving the following upper bound for the threshold of completeness of positive nth powers:

$$(b-1)2^{n-1}\big(r + \tfrac{2}{3}(b-1)(2^{2n}-1) + 2(b-2)\big)^n - 2a + ab,$$

where $a = n!2^{n^2}, b = 2^{n^3}a^{n-1}, r = 2^{n^2-n}a$ [4].

For odd exponents, not being careful about stating that only positive powers are permitted can lead to the misunderstanding that we can include negative powers, such as $(-2)^3$. Permitting both addition and subtraction of powers is not an interesting generalization of the threshold of completeness problem. We can see this with the following method that uses Prouhet's solution to the Tarry-Escott problem [19]. Consider the 2^n consecutive powers $u^n, (u-1)^n, \ldots, (u-2^n+1)^n$ added and subtracted with the signs of the powers in a Prouhet-Thue-Morse sequence $(+--+-++--++-+--+\ldots)$ [9, 16, 17, 6], A010060 in the *On-Line Encyclopedia of Integer Sequences* [13]. Each power is a polynomial in u whose highest degree term is $\pm u^n$. When we take the powers in 2^{n-1} adjacent pairs, the pairs are finite differences with a spacing of ± 1, the u^n terms cancel, and we are left with polynomials in u whose highest degree term is $\pm nu^{n-1}$. When we take these polynomials in 2^{n-2} pairs, each pair is a second order finite difference with a spacing of ± 2 and is a polynomial whose highest degree term is $\pm 2n(n-1)u^{n-2}$. Each step of pairing up the polynomials produces higher order finite differences (using spacings equal to plus or minus a power of 2) that reduce the polynomial degree by 1, and all resulting polynomials have the same highest degree term except for the sign. The final sum is the constant $c = n!2^{n(n-1)/2}$, independent of u. This means that by varying u, we get an infinite supply of sums equal to c, each using 2^n consecutive powers (from non-overlapping ranges). Now consider any power of the form $(vc+1)^n$ for arbitrary v. Such powers are congruent to 1 (mod c), and we can subtract enough sums equal to c to get a sum of 1. This means we now have an infinite supply of sums equal to 1. With these we can form any sum that is positive, negative, or zero.

n	Threshold of Completeness for nth powers	Reference
1	0	
2	128	[14]
3	12,758	[3, 1]
4	5,134,240	[5]
5	67,898,771	[7]
6	11,146,309,947	[2]
7	766,834,015,734	[13]
8	4,968,618,780,985,762	this work
9	$\geq$ 155,581,444,629,727,232	⋮
10	$\geq$ 13,130,572,287,326,740,862	
11	$\geq$ 1,641,961,088,738,387,417,337	
12	$\geq$ 555,664,230,155,437,340,241,901	
13	$\geq$ 132,999,999,685,872,578,648,583,148	
14	$\geq$ 13,799,996,074,582,156,266,244,947,666	
15	$\geq$ 2,099,999,963,240,136,765,496,329,587,024	
16	$\geq$ 999,999,988,996,135,865,287,858,019,238,512,392	

Table 1: Thresholds of completeness.

According to the *On-Line Encyclopedia of Integer Sequences* (A001661), thresholds of completeness are known for sets of positive nth powers for exponents $n = 1, 2, \ldots, 7$, with the $n = 7$ case solved by Johnson in 2010 [13]. Our contribution is the threshold of completeness for 8th powers and lower bounds for some higher exponents (see Table 1). We also describe the mathematical methods we developed to compute thresholds of completeness in enough detail that others can reproduce and extend our results.

2 Finding thresholds of completeness by finite search

When we say that a threshold of completeness has a particular value, in part we are asserting that all larger values are summable (see Definition 2). Fortunately, we can prove this with a finite search.

Definition 1. Let $S(x^n) = \{1^n, 2^n, \ldots\}$, and let $S_k(x^n) = \{1^n, 2^n, \ldots, k^n\}$.

Definition 2. The threshold of completeness for nth powers, $T(S(x^n))$, is the largest integer that is not the sum of one or more distinct elements of the set $S(x^n)$.

According to Porubský [8], Theorem 3 below began with Richert proving that every integer greater than 6 is the sum of distinct primes [10], and Sierpiński extending it to more general purposes [12]. The version below is adapted for finding thresholds of completeness.

Theorem 3. *Given $n, k \in \mathbb{Z}^+$ and $t \in \mathbb{Z}$ such that*

(1) $t+1$ to $t+(k+1)^n$ (inclusive) are summable with the finite set $S_k(x^n)$, and

(2) $(k+2)^n \leq 2(k+1)^n$,

then the threshold of completeness $T(S(x^n)) \leq t$.

Proof. Let $a \geq 0$. Rewrite condition (2) as $(k+2)/(k+1) \leq 2^{1/n}$. Then

$$\frac{k+a+2}{k+a+1} = \frac{k+2}{k+1} - \frac{a}{(k+a+1)(k+1)} \leq \frac{k+2}{k+1} \leq 2^{1/n}.$$

So, if condition (2) holds for k, it also holds if we replace k with any integer larger than k. Next we use induction. Consider the statement that all integers from $t+1$ to $t+(b+1)^n$ (inclusive) are summable with the finite set $S_b(x^n) = \{1^n, 2^n, \ldots, b^n\}$. Assume the statement is true for some $b \geq k$. We can then add $(b+1)^n$ to each solution from $t+1$ to $t+(b+1)^n$ so that all integers from $t+1$ to $t+2(b+1)^n$ are summable using powers no larger than $(b+1)^n$. Because $(b+2)^n \leq 2(b+1)^n$, we now have that all integers from $t+1$ to $t+(b+2)^n$ are summable using the set $S_{b+1}(x^n)$. This proves that if the statement is true for b, then it is true for $b+1$. By condition (2), the statement is true for $b = k$. This completes the proof by induction that the statement is true for all $b \geq k$. So all integers from $t+1$ to $t+(b+1)^n$ are summable from $S(x^n)$ for b growing without bound. Thus, all integers greater than t are summable, and the threshold of completeness $T(S(x^n)) \leq t$, as required. □

Theorem 3 does not directly give us the threshold of completeness, but once we have t and k that satisfy its conditions, the problem is reduced to a finite search through integers less than or equal to t for the largest integer that is not summable. We could add a third condition to this theorem that t be non-summable so that it is equal to the threshold of completeness. However, we found that adding this condition is inefficient for exponents 4 and 6 because it requires a larger value for k (see the discussion of Table 2).

3 Exponents up to 6

We describe a simple and fast sieve-like method of finding the threshold of completeness for small exponents to introduce concepts that will be useful for larger exponents. We begin with a memory consisting of an array of bits $b_0, b_1, \ldots, b_\mu$ for some upper limit μ. If $\mu \geq t+(k+1)^n$, where n, t, and k are as defined in Theorem 3, then the algorithm described below will succeed in finding $T(S(x^n))$. To start, set $b_0 = 1$, and the remaining bits to 0. In this state, we are indicating that using no elements of $S(x^n)$ at all, the only sum we can form is 0. Although Definition 2 requires a valid sum to include at least one power, it is convenient to consider 0 to be the sum of the null set of powers for this computation. Let us consider the case of squares using $\mu = 255$. The starting state of the memory is $b_{255} = \cdots = b_1 = 0$, and $b_0 = 1$. Next we make use of Algorithm 5 for adding a power y to every existing sum in the memory.

Definition 4. We use $\wedge$ as the bitwise AND operation, and $\vee$ as the bitwise OR operation. Let x, y, z be nonnegative integers with binary representations $x = \sum_{i=0}^{\infty} x_i 2^i$, $y = \sum_{i=0}^{\infty} y_i 2^i$, $z = \sum_{i=0}^{\infty} z_i 2^i$, and all $x_i, y_i, z_i \in \{0, 1\}$. We say that z is the bitwise AND of x and y and write $z = x \wedge y$ if for all $i \in \mathbb{Z}_0^+$, $z_i = x_i$ AND y_i. Similarly, we say that z is the bitwise OR of x and y and write $z = x \vee y$ if for all $i \in \mathbb{Z}_0^+$, $z_i = x_i$ OR y_i.

Algorithm 5. *Add New Power to Memory.* Take a power y and memory limit μ, and add to the memory y more than every sum already represented in the memory. Given the memory bits $b_0, b_1, \ldots, b_\mu$, we go through the b_i from $i = \mu$ down to y, and perform $b_i \leftarrow b_i$ OR b_{i-y}. If we think of the memory as a single large integer $W = \sum_{i=0}^{\mu} b_i 2^i$, this operation is simply $W \leftarrow W \vee (2^y W \bmod 2^{\mu+1})$ (see Definition 4).

For the first three times we use Algorithm 5 below, we show only $b_{14}, \ldots, b_0$ because the higher order bits are all zeros after these steps. Now we use Algorithm 5 to add the power $y = 1^2$ to the memory. The result is

$$b_{14} \text{ to } b_0 : 000000000000011,$$

which means that the possible sums are 0 and 1. Next we add in 2^2 to get

$$b_{14} \text{ to } b_0 : 000000000110011,$$

corresponding to sums 0, 1, 4, 5. After the step with 3^2, we get

$$b_{14} \text{ to } b_0 : 110011000110011.$$

After the $4^2, 5^2, \ldots, 10^2$ steps, we have

$$\begin{aligned}
b_{63} \text{ to } b_0 &: 11101111111111100110011111111100011001100011001101100110001100 11 \\
b_{127} \text{ to } b_{64} &: 11111111111111011101111111111101110111111111111111011101111011 11 \\
b_{191} \text{ to } b_{128} &: 11111111111111111111111111111111111111111111111111111111111111 10 \\
b_{255} \text{ to } b_{192} &: 11111111111111111111111111111111111111111111111111111111111111 11.
\end{aligned}$$

Now $b_{128} = 0$ and all the higher order bits are 1. The conditions of Theorem 3 are met for $n = 2$, $t = 128$, and $k = 10$. All that remains is to find the largest integer less than or equal to 128 that is not summable. We proceed with the 11^2 step with Algorithm 5 because $11^2 \leq 128$, and see that it does not change b_{128}. Thus $T(S(x^2)) = 128$.

Table 2 shows results for powers up to 6 using this method on a laptop. We see that $T(S(x^n)) < t$ for exponents 4 and 6. If we add the requirement that $t = T(S(x^n))$ to Theorem 3, execution times increase for the exponent 4 and 6 cases, because we have to increase k to 38 and 46, respectively. (All execution times in this paper are measured using a single thread on a 12th Generation Intel(R) Core(TM) i9-12900HK running at 2.50 GHz. For memory sizes, we use kB = 1024 bytes, MB = 1024 kB, GB = 1024 MB, and TB = 1024 GB.)

n	$T(S(x^n))$	t	k	Memory Size	Run Time
1	0	0	1	2 bits	< 1 μs
2	128	128	10	250 bits (32 bytes)	1.3 μs
3	12,758	12,758	20	22,020 bits (3 kB)	27 μs
4	5,134,240	6,211,329	37	8,296,466 bits (1.0 MB)	2.4 ms
5	67,898,771	67,898,771	33	113,334,196 bits (14 MB)	28 ms
6	11,146,309,947	12,840,617,485	42	19,161,980,535 bits (2.3 GB)	7.7 s

Table 2: Results for small exponents.

Unfortunately, using this simple and fast method would require 152 GB for 7th powers and 776 TB for 8th powers, which is well beyond the memory capacity of a typical laptop. It would be possible to use the laptop's solid state drive for 7th powers, and a modern supercomputer could handle 8th powers, but we chose to find ways to trade extra execution time for reduced memory requirements.

4 Reducing memory requirements

Here we describe a way to reduce the size of the memory required to find the threshold of completeness for exponents $n = 7$ and higher. We begin with an inefficient method and improve it in a sequence of steps, finishing with a complete algorithm.

4.1 Index searching

To continue, we need to introduce the concept of an *index*.

Definition 6. An *index* is a nonnegative integer representing a finite set of positive nth powers, where the exponent n is explicitly stated or implied by context. Let $s_i = 1$ when power $(i+1)^n$ is in the set, and $s_i = 0$ otherwise. Then the set's index is $I = \sum_{i=0}^{\infty} s_i 2^i$. The sum of the elements in the set of powers represented by index I is $\sum_{i=0}^{\infty} s_i (i+1)^n$. Any summable number has at least one corresponding set of powers that contribute to that sum. A given sum may have more than one index. For example $25 = 5^2 = 4^2 + 3^2$. The set $\{5^2\}$ has index $2^{5-1} = 16$, and the set $\{3^2, 4^2\}$ has index $2^{3-1} + 2^{4-1} = 12$, but the sum of powers corresponding to both indexes is 25.

Let $b_0, \ldots, b_\mu$ be the available bits of memory for some upper limit μ. In Section 3, we iterated Algorithm 5 to set $b_i = 1$ when i is a summable number. An alternative way to fill the memory begins with setting all memory bits $b_0, \ldots, b_\mu$ to zero. Then for each index $I = 0, \ldots, 2^k - 1$, where $k = \lfloor \mu^{1/n} \rfloor$, compute the sum v of the nth powers in the set represented by I, and if $v \leq \mu$, set $b_v = 1$. This method is far less efficient than the method from Section 3 and does not reduce memory requirements, but we will improve it below.

4.2 Separating sums by congruence class

In Section 4.1, at the end of computation, a memory bit b_v, $0 \le v \le \mu$, was equal to 0 if v is not summable, and 1 if v is summable. We got this result with a single large computation. To reduce memory requirements (or to make use of the $\mu + 1$ available bits of memory for determining the summability of integers larger than μ), we separate sums into congruence classes mod 2^{gn} for some $g \in \mathbb{Z}^+$. Let $L = (\mu + 1)2^{gn} - 1$. With this new approach, we repeat the following subcomputation for each $m = 0, 1, \ldots, 2^{gn} - 1$. Set all memory bits b_i to zero. For each index $I = 0, \ldots, 2^k - 1$, where $k = \lfloor L^{1/n} \rfloor$, compute the sum v of the nth powers in the set represented by I (see Definition 6), and if $v \le L$ and $v \equiv m \pmod{2^{gn}}$, set $b_{(v-m)2^{-gn}} = 1$. At the end of each subcomputation, a memory bit b_i is equal to 0 if $i2^{gn} + m$ is not summable, and 1 if it is summable.

For a given memory limit μ, the methods of Section 4.1 could only determine the summability of numbers up to μ, but this new approach can determine summability of numbers up to limit $L = (\mu+1)2^{gn} - 1$. This algorithm is even slower than the one in Section 4.1, but at least the memory requirements are reduced for a given limit L. The advantage of separating sums by congruence class will become clearer in Section 4.3.

4.3 Separating even and odd powers

To discuss this optimization, it is useful to introduce the concept of a *subindex*.

Definition 7. A *subindex* is a nonnegative integer representing a finite set of positive odd nth powers, where the exponent n is explicitly stated or implied by context. Let $s_i = 1$ when power $(2i+1)^n$ is in the set, and $s_i = 0$ otherwise. Then the set's subindex is $I = \sum_{i=0}^{\infty} s_i 2^i$.

Definition 8. Let $\sigma_n(I)$ be the sum of the elements in the set of odd powers represented by subindex I: $\sigma_n(I) = \sum_{i=0}^{\infty} s_i(2i+1)^n$, where $I = \sum_{i=0}^{\infty} s_i 2^i$, and $s_i \in \{0, 1\}$ for all $i \in \mathbb{Z}_0^+$.

This optimization starts with the simple but powerful observation that all even powers are divisible by 2^n so that only odd powers determine a sum's congruence class mod 2^n. If we are only looking for sums congruent to $m \pmod{2^n}$, then we need consider only subsets of the odd powers whose sum is congruent to $m \pmod{2^n}$. Then we can add any combination of even powers we like without changing the congruence class of the sum. So, where the algorithm in Section 4.2 with $g = 1$ searched through indexes $0, \ldots, 2^k - 1$ to cover all subset sums of $S_k(x^n)$, we now cover all the subset sums of just the odd powers less than or equal to k^n by searching through the subindexes $I = 0, \ldots, 2^{\lceil k/2 \rceil} - 1$. It is only after finding all subindexes I such that $\sigma_n(I) \equiv m \pmod{2^n}$ and setting memory bit $b_{(\sigma_n(I)-m)2^{-gn}} = 1$ that were turn our attention to the even powers. Because the sums represented by the memory bits are spaced 2^n apart, when we use Algorithm 5 with $y = p^n$, this has the effect of adding even power $(2p)^n$ to all the sums in the memory. So, we run Algorithm 5 for each $y = 1^n, \ldots, \lfloor k/2 \rfloor^n$ to cover all the even powers less than or equal to k^n.

So far, we have used the idea in Section 4.2 with $g = 1$, but we can go further. Suppose that we seek sums of powers congruent to $m \pmod{2^{2n}}$ instead of just $m \pmod{2^n}$. Now, in a first stage, we enumerate all subindexes $I = 0, \ldots, 2^{\lceil k/2 \rceil} - 1$ and for each I, compute its sum of odd powers $s = \sigma_n(I)$, and if $s \equiv m \pmod{2^n}$, then carry $\lfloor s/2^n \rfloor$ and $\lfloor m/2^n \rfloor$ to a nearly identical second stage that uses $\lfloor k/2 \rfloor$ in place of k. The first stage took care of the least significant n bits of m, and in the second stage we can think of all sums as divided by 2^n. Again, we go through the subindexes, this time looking for sums of powers congruent to $\lfloor m/2^n \rfloor \pmod{2^n}$. A difference between the first and second stage is that we add the offset $\lfloor s/2^n \rfloor$ to each sum in the second stage. In a final step where we think of all sums as divided by 2^{2m}, we use Algorithm 5 to add powers $1^n, 2^n, \ldots, \lfloor k/4 \rfloor^n$ to the memory. If we step back and do not think of sums as divided by either 2^n or 2^{2n}, the first stage found sums of odd powers, the second stage contributed sums of powers x^n, where $x \equiv 2 \pmod 4$, and the final step contributed sums of powers x^n, where $x \equiv 0 \pmod 4$. See Algorithms 12 and 13 for a more precise description of this process. There is no reason why we have to stop there. We can increase g further to reduce memory requirements at the cost of more execution time.

4.4 Efficiently avoiding sums of powers that are too large

In Section 4.3, enumerating all the sums of odd powers in $S_k(x^n)$ involved counting through the subindexes $I = 0, \ldots, 2^{\lceil k/2 \rceil} - 1$. We are only interested in sums $\sigma_n(I) \leq L = (\mu+1)2^{gn} - 1$, where $\mu + 1$ is the number of memory bits available. However, some of the sums may be larger than L. In fact, it is common for many subindexes in a row to produce sums $\sigma_n(I)$ that are larger than L. This slows down the algorithm considerably. Given a subindex I where $\sigma_n(I) > L$, we would like an efficient way to find the next larger subindex I' such that $\sigma_n(I') \leq L$. Theorem 9 is useful here.

Theorem 9.

(1) Consider a subindex I that has at least two 1s in its binary representation. Let 2^u be the weight of the least significant 1 in the binary representation of I, and let 2^v be the weight of the second least significant 1 in the binary representation of I. Then no subindex $I' \in (I, I - 2^u + 2^v)$ is such that $\sigma_n(I') \leq \sigma_n(I)$.

(2) Consider a subindex I that has at most one 1 in its binary representation. Then no subindex $I' > I$ is such that $\sigma_n(I') \leq \sigma_n(I)$.

Proof.

(1) Let $j = I - 2^v - 2^u$, where $v > u$. Then all 1s in the binary representation of j have weight larger than 2^v, and $(I, I - 2^u + 2^v) = (j + 2^v + 2^u, j + 2^v + 2^v)$. If $I' \in (I, I - 2^u + 2^v)$, then $I' = j + 2^v + b$, where $b \in (2^u, 2^v)$. Because $I = j + 2^v + 2^u$, the binary representations of I and I' differ only in bits of weight less than 2^v. For I, the only contribution of bits of weight less that 2^v to the sum $\sigma_n(I)$ is $(2u + 1)^n$. For

I', the contribution of bits of weight less that 2^v to $\sigma_n(I')$ is all the bits in b. But the binary representation of b must either have a 1 of weight 2^u along with at least one other 1, or b must have a 1 with weight larger than 2^u. In either case, $\sigma_n(b) > (2u+1)^n$, and thus $\sigma_n(I') > \sigma_n(I)$ for all $I' \in (I, I - 2^u + 2^v)$.

(2) If the binary representation of I has fewer than two 1s, then either $I = 0$, in which case $\sigma_n(I) = 0$ and the statement is trivially true, or $I = 2^u$ for some u. The binary representation of any subindex $I' > I$ must either have a 1 of weight 2^u along with at least one other 1, or it must have a 1 with weight larger than 2^u. In either case, $\sigma_n(I') > \sigma_n(I)$.

□

Suppose that the least significant 1 in the binary representation of subindex I has weight 2^u, and the second least significant 1 has weight 2^v (we will deal with the possibility that I does not have two 1s below). By Theorem 9, if $I' \in (I, I - 2^u + 2^v)$, then $\sigma_n(I') > \sigma_n(I)$. An interesting trick from Warren's useful book *Hacker's Delight* [18, p. 11] for isolating the least significant 1 in the binary representation of I is to calculate $I \wedge (2^W - I)$, where W is any positive integer such that $2^W > I$, and $\wedge$ refers to a bitwise AND operation (see Definition 4). Because computers naturally work modulo 2^{32} or 2^{64} for unsigned integers, performing $I \wedge (2^W - I)$ is the same as $I \wedge -I$ for any I that fits in the computer's word size. So we will just write this computation as $I \wedge -I$. Because $I \wedge -I = 2^u$, and $(I - 2^u) \wedge -(I - 2^u) = 2^v$, we can compute $I - 2^u + 2^v$ as follows:

$$I \leftarrow I - (I \wedge -I) \quad \text{followed by} \quad I \leftarrow I + (I \wedge -I).$$

If the result is $I = 0$ because the binary representation of I started with fewer than two 1s, this means there are no remaining subindexes $I' > I$ such that $\sigma_n(I') \leq \sigma_n(I)$. So, when we go through the subindexes I from 0 to $2^{\lceil k/2 \rceil} - 1$ as described in Section 4.3, and we encounter an I such that $\sigma_n(I) > L = (\mu + 1)2^{gn} - 1$, we use the two *Hacker's Delight* steps above to skip over subindexes I' where $\sigma_n(I') > L$.

4.5 Pre-computing sums sorted by congruence class

In Section 4.3, when enumerating all the sums of subsets of odd powers in $S_k(x^n)$ by counting through the subindexes from 0 to $2^{\lceil k/2 \rceil} - 1$, we seek sums congruent to $m \pmod{2^n}$. A more efficient version of this process begins with precomputing lists of some of the sums of subsets of odd powers. For some constant p, we precompute the sums $s_J = \sigma_n(J)$ for subindexes $J = 0, \ldots, 2^p - 1$, making a separate list based on each sum's congruence class mod 2^n, and we sort the sums s_J within each list from smallest to largest. With each sum, we also store its corresponding subindex J. These precomputed lists allow us to speed up the computation as shown in Algorithm 12.

4.6 Reducing the number of odd powers

Here we describe an optimization that involves ignoring some of the odd powers. This is useful for exponent $n = 8$. We begin with an observation about even exponents.

If we add up an odd number of odd powers we get an odd sum. So, if $\sigma_n(I)$ is odd then subindex I must have odd Hamming weight. (We use "Hamming weight" to mean the number of 1 bits in the binary representation of a nonnegative integer.) Similarly, if $\sigma_n(I)$ is even then subindex I must have even Hamming weight. We can extend this idea further with Lemma 10 and Theorem 11.

Lemma 10. *Suppose $a \in \mathbb{Z}$ and $u \in \mathbb{Z}^+$. Then $(2a+1)^{2^u} \equiv 1 \pmod{2^{u+2}}$.*

Proof. We proceed by induction. Consider $u = 1$.

$$(2a+1)^2 = 4a^2 + 4a + 1 = 8\tbinom{a+1}{2} + 1 \equiv 1 \pmod 8.$$

Thus this lemma is true for $u = 1$. Now assume it is true for $u = v$, for some $v > 0$. So $(2a+1)^{2^v} = 2^{v+2}b + 1$, for some integer b. Then

$$(2a+1)^{2^{v+1}} = (2^{v+2}b+1)^2 = 2^{2v+4}b^2 + 2^{v+3}b + 1 \equiv 1 \pmod{2^{v+3}}.$$

Thus, if this lemma is true for $u = v$, it is also true for $u = v + 1$. This completes the proof by induction. □

Theorem 11. *Consider an exponent n that is divisible by 2^u, $u > 0$. If a subindex I has Hamming weight h, then $\sigma_n(I) \equiv h \pmod{2^{u+2}}$.*

Proof. A consequence of Lemma 10 is that all odd powers for exponent n are congruent to 1 (mod 2^{u+2}). When subindex I has Hamming weight h, $\sigma_n(I)$ is the sum of h odd powers. Thus $\sigma_n(I) \equiv h \pmod{2^{u+2}}$. □

For exponent $n = 8$, let I be a subindex with Hamming weight h and let $q = \sigma_8(I)$. Then by Theorem 11, $h \equiv q \pmod{32}$. So, when we are enumerating subindexes I looking for those such that $\sigma_8(I) \equiv m \pmod{2^{8g}}$ for some $g \geq 1$, we need only consider subindexes with Hamming weight congruent to m (mod 32). However, the number of subindexes with Hamming weight congruent to m (mod 32) varies considerably with m. Consider the case of $k = 80$ so that there are only 40 odd powers. For $m = 4$, a subindex must have Hamming weight 4 or 36. There are only $\binom{40}{4} + \binom{40}{36} = 182{,}780$ such subindexes available. In contrast, for $m = 20$, there are $\binom{40}{20} = 137{,}846{,}528{,}820$ subindexes available. This increase by a factor of over 750,000 makes a huge difference in how long it takes to enumerate all the subindexes. Additionally, the extra computation is mostly wasted because in Algorithm 12 below, we are just setting the same memory bits to 1 over and over again. To avoid this waste, we eliminated some of the larger odd powers from our search. For $k = 80$, we would normally consider the odd powers $1^8, 3^8, \ldots, 79^8$. Instead, we defined k' to be an upper limit on the number of odd powers used. So, when computing subset sums of $S_k(x^8)$, we did not include

the powers $(2k'+1)^8, (2k'+3)^8, \ldots, (2\lceil k/2\rceil - 1)^8$ in any of the subsets. We determined a suitable k' for each congruence class $m \pmod{32}$ experimentally. If at any point k' was not large enough to satisfy the conditions of Theorem 3, we simply increased k' (without letting $2k'-1$ exceed k) and tried again. See Section 6 for more details on this optimization.

Another possible use for this observation about the Hamming weight of subindexes would be to directly search through just those subindexes with the correct Hamming weight instead of just trying every subindex. This idea works well for exponent $n = 16$ as we will see in Section 9, but for exponent $n = 8$, it was more efficient to use pre-computed sums of odd powers as described in Section 4.5.

4.7 Efficient method for checking one congruence class

The goal for a given exponent n is to find integers t and k that will satisfy the requirements of Theorem 3. We do not know in advance which values of t and k will work; finding them is an iterative process. We make guesses and perform computations to reveal whether we need to change these guesses. Combining the efficiency improvements in the subsections above gives the following algorithms that we can use in this iterative process. Algorithm 12 performs the stages of computing sums of odd powers. One use for this algorithm is to check the range from $t+1$ to $t+(k+1)^n$ for numbers not summable with $S_k(x^n)$. If a number is not summable, then we have a new larger candidate for t. Another use is to find numbers smaller than t that are not summable with $S(x^n)$. In this case, we use $\lfloor t^{1/n} \rfloor$ in place of k so that we are considering all powers not larger than t. Algorithm 13 uses Algorithm 12 to determine whether we need a larger value for t to satisfy Theorem 3 and to determine if there exists a larger candidate c for the threshold of completeness.

Algorithm 12. *Calculate Sums of Odd Powers for One Congruence Class.* The inputs are a congruence class $m \bmod 2^{gn}$, an exponent n, the number of remaining stages g of adding sums of odd powers, an offset f to be added to each sum, the number k which sets k^n to be the largest power used in sums of powers, the number k' which sets $(2k'-1)^n$ to be the largest odd power used in the sums, and the upper limit L on the size of sums sought. The output is a memory consisting of bits $b_0, \ldots, b_\mu$, where $\mu = \lfloor (L-m)/2^{gn} \rfloor$. At the end of the algorithm, $b_a = 1$ if $a2^{gn} + m$ is summable using odd powers $1^n, 3^n, \ldots, (2k'-1)^n$ and all even powers less than or equal to k^n except for those of the form x^n where $x \equiv 0 \pmod{2^g}$. Bit $b_a = 0$ otherwise. For each $d = 1, 2, \ldots, p$, we use a precomputed table of pairs (s_j, j), where j is a d-bit subindex, and $s_j = \sigma_n(j)$ is a sum of odd powers in $S_{2d}(x^n)$. Each table is separated into distinct lists by each sum's congruence class mod 2^n. Each list is sorted from smallest to largest sum.

for $a = 0, 1, \ldots, \mu$ **do** $b_a \leftarrow 0$ **end for**
maxq $\leftarrow L - f$
$i \leftarrow 0$
$q \leftarrow 0$
do

use precomputed table of sums of odd powers with subindexes $\min(p, k')$ bits long
use precomputed list of sums of odd powers congruent to $m - q - f \pmod{2^n}$
if $g > 1$ **then**
for each precomputed pair (s_j, j) such that $s_j \le \text{maxq} - q$ **do**
recursively call Algorithm 12 with inputs
$(\lfloor m/2^n \rfloor,\ n,\ g-1,\ \lfloor (q+f+s_j)/2^n \rfloor,\ \lfloor k/2 \rfloor,\ \lceil \lfloor k/2 \rfloor /2 \rceil,\ \lfloor L/2^n \rfloor)$
end for
else
for each precomputed pair (s_j, j) such that $s_j \le \text{maxq} - q$ **do**
$b_{\lfloor (q+f+s_j)/2^n \rfloor} \leftarrow 1$
end for
end if
$i \leftarrow i + 2^p$
$q \leftarrow \sigma_n(i)$
while $q > \text{maxq}$ and $i \ne 0$ and $i < 2^{k'}$ **do**
$i \leftarrow i - (i \wedge (-i))$
$i \leftarrow i + (i \wedge (-i))$
$q \leftarrow \sigma_n(i)$
end while
while $i \ne 0$ and $i < 2^{k'}$

Algorithm 13. *Check One Congruence Class.* The inputs are a congruence class m mod 2^{gn}, an exponent n, the current largest candidate c for the threshold of completeness, the current value for t being used to satisfy Theorem 3, the number of stages g of adding sums of odd powers, limits k and k' ($k' \le \lceil k/2 \rceil$) on the size of powers used in the sums, and the upper limit L on the size of sums sought. For odd powers, we do not use any powers larger than $(2k'-1)^n$. For even powers, we do not use any powers larger than k^n. We use a memory consisting of bits b_i, $i = 0, 1, \ldots, \mu$, where $\mu = \lfloor (L-m)/2^{gn} \rfloor$. To satisfy the requirements of Theorem 3, we must have $L \ge t + (k+1)^n$. The output is a boolean value along with possible updates to the values of c, t, k, and L. The output is (true) if the conditions of Theorem 3 are met for congruence class $m \pmod{2^{gn}}$, or if we are able to update c or t to meet these conditions. Otherwise, the output is (false) and either k has been increased or L has been increased (in which case we must run this algorithm again with these new inputs). When the output is (false), it is necessary to start the process of checking all congruence classes $m \pmod{2^{gn}}$ over again. When the memory size must increase, the suggested size may be larger than necessary, because k may have grown too large. In practice, when the memory size had to increase, we often chose the new size and the starting value of k manually.

$z \leftarrow L$
do
call Algorithm 12 with inputs (m, n, g, 0, k, k', z)
for $j = 1, 2, \ldots, \lfloor k/2^g \rfloor$ **do**

call Algorithm 5 with power j^n
end for
find the largest $a \in [0, \mu]$ such that $b_a = 0$
if there is no such a **then return** (true) **end if**
$z \leftarrow m + a2^{gn} - 1$
if $z < c$ **then return** (true) **end if**
$k' \leftarrow k' + 1$
while $k' \leq \lceil k/2 \rceil$
$v \leftarrow 0$
while $(v < z)$ and $(\exists$ minimum $a \in (\lfloor (t-m)/2^{gn} \rfloor, \mu]$ such that $b_a = 0)$ **do**
$v \leftarrow m + a2^{gn}$
if $(v - t) > (k+1)^n$ **then** $z \leftarrow v - 1$ **else** $t \leftarrow v$ **end if**
end while
call Algorithm 12 with inputs (m, n, g, 0, $\lfloor t^{1/n} \rfloor$, $\lceil \lfloor t^{1/n} \rfloor / 2 \rceil$, t)
for $j = 1, 2, \ldots, \lfloor t^{1/n}/2^g \rfloor$ **do**
call Algorithm 5 with power j^n
end for
find the largest $a \in [0, \mu]$ such that $b_a = 0$
$v \leftarrow m + a2^{gn}$
if $v > c$ **then**
$c \leftarrow v$
if $t < c$ **then** $t \leftarrow c$ **end if**
if $(L - c) < (k+1)^n$ **then** $L \leftarrow c + (k+1)^n$; **return** (false) **end if**
end if
if $(L - t) < (k+1)^n$ **then** $k \leftarrow k + 1$; $t \leftarrow c$; **return** (false) **end if**
if $(z - t) < (k+1)^n$ **then return** (false) **end if**
return (true)

4.8 Other observations

Another observation for $n = 8$ is that all powers are congruent to 0, 1, or 16 modulo 17. For each exponent, we can come up with many other similar facts for different moduli. There may be some way to combine several such constraints on powers to speed up the search for the threshold of completeness, but we do not see a way to use these ideas to further improve on the optimizations described above.

5 Exponent 7

We used Algorithm 13 to find $T(S(x^7))$. We began by calling this algorithm with $m = 0$, $n = 7$, $c = 0$, $t = 0$, $g = 1$, $k = 40$, $k' = 20$, and $L = 10^{12}$. Because the exponent is odd, there

is nothing to be gained by reducing the number of odd powers as explained in Section 4.6, so we always used $k' = \lceil k/2 \rceil$. For the first 24 calls, the algorithm returned (false) and increased one or more of t, k, and L, so we kept starting over at $m = 0$. After that, the algorithm would return (true) for several congruence classes mod 2^7 before failing at some point and forcing us to start over again. Eventually, Algorithm 13 succeeded for all $m = 0, 1, \ldots, 2^7 - 1$ using t = 865,130,689,840 and $k = 45$ to give $c = T(S(x^7))$ = 766,834,015,734. The whole process took 19 minutes. If we reduce L to the bare minimum required, the time required to confirm this result is 7.7 minutes using 1.2 GB of memory.

6 Exponent 8

The work to this point has been a warm-up for the main contribution, which is the threshold of completeness for exponent $n = 8$. We found this threshold in four steps. First we did an initial exploration to find roughly the right sizes of inputs to Algorithm 13. Next we searched for larger values of c, t, and k. Then we found suitable values for k' to minimize computation time. Finally, we did the full computation to get $T(S(x^8))$.

Before the initial exploration, it was clear that using only $g = 1$ stage of adding odd powers would require too much memory. In Section 4.6, we saw that a subindex must have Hamming weight $h \equiv m \pmod{32}$. To create a sum of odd powers congruent to 31 (mod 32), the set of odd powers contributing to this sum must contain at least 31 odd powers, and the subindex for this set must have a Hamming weight of at least 31. So, k' must be at least 31, and k must be at least 61. Theorem 3 requires $L \geq t + (k+1)^8$, and thus $L \geq 62^8$. The number of memory bits required is $\lceil L/2^{gn} \rceil$. With $g = 1$, this requires a 100 GB memory, which is beyond the resources we had available. Using $g = 2$ reduces this to 400 MB, and leaves room for expansion as t and k increase.

In the initial exploration, we focused on the congruence class $m \pmod{2^{16}}$ likely to give few subindexes. This has two advantages. The first is that is takes less computation time to go through these subindexes. The second is that it makes intuitive sense that in such a case we get our best chance of finding a large candidate for the threshold of completeness c, and a large value for t to use for Threorem 3. There is no guarantee that this intuition is correct, but the only cost if we are wrong is greater execution time; it will not lead to an incorrect final result. Given k' and m, the number of subindexes with Hamming weight congruent to $m \pmod{32}$ is

$$Q_{32}(k', m) = \sum_{\substack{i \equiv m \pmod{32} \\ 0 \leq i \leq k'}} \binom{k'}{i}.$$

This is a minimum when $m \equiv \lfloor k'/2 \rfloor - 16 \pmod{32}$. When k' is odd, we could have used $m \equiv \lceil k'/2 \rceil - 16 \pmod{32}$, because it gives the same sum (the terms are identical but in reverse order). However, we prefer the first minimum because it has more high Hamming weight subindexes I whose sums of odd powers $\sigma_n(I)$ tend to be larger. When we add in even powers, larger sums of odd powers have a greater tendency to produce irrelevant total

sums that exceed L. We began our initial exploration with $m = 0$, $n = 8$, $c = 0$, $t = 0$, $g = 2$, $k = 64$, $k' = 32$, and $L = 10^{14}$. As Algorithm 13 changed c, t, k, and L, we changed k' to equal $\lceil k/2 \rceil$ and m to equal $\lfloor k'/2 \rfloor - 16$. After 5.5 hours, call number 172 returned (true) for inputs $m = 4$, $n = 8$, $c = t =$ 4,739,704,992,497,666, $g = 2$, $k = 79$, $k' = 40$, and $L =$ 6,417,426,592,497,666.

In the next step, we explored the most promising congruence classes $m \pmod{2^{16}}$ to give a value of c closest to the actual threshold of completeness, and values of t and k closest to those needed for Theorem 3. There are 2^{11} such congruence classes in each mod 32 congruence class. We judged congruence classes 4 and 3 (mod 32) to be the most promising based on the number of subindexes available for $k = 79$ and $k' = 40$. First, we checked $m = 4, 36, \ldots, 2^{16} - 28$, but Algorithm 13 returned (true) in every case. Then we checked $m = 3, 35, \ldots, 2^{16} - 29$ to reveal that we needed to use $k = 80$, $t =$ 5,784,315,489,954,275, and $L =$ 7,637,335,678,806,116. Although we did not know it at the time, these turned out to be the final values needed for use with Theorem 3.

The next step was to find the right k' values for minimizing the execution time of the final step. Let k'_m be the smallest k' such that Algorithm 13 returns (true) for congruence class $m \pmod{2^{16}}$. We found $k'_0, k'_1, \ldots, k'_{31}$:

36,37,39,40,38,32,26,22,21,21,20,21,21,21,22,22,23,24,25,25,26,27,28,29,29,30,31,32,33,33,34,35.

We used these values in the final step (below) to speed up the computation. For a given $m < 2^{16}$, let $a = m \bmod 32$. We assumed that a reasonable starting guess for k'_m is k'_a. In some cases, k'_a was not large enough, and Algorithm 13 had to increase it. Let k^*_a be the smallest value of k' such that Algorithm 13 returns (true) for every congruence class $m = a, a + 32, \ldots, a + 2^{16} - 32 \pmod{2^{16}}$. Here is the final list of $k^*_0, k^*_1, \ldots, k^*_{31}$:

36,37,40,40,39,32,27,23,22,21,21,21,21,22,22,23,23,24,25,26,26,27,28,29,30,30,31,32,33,33,34,35.

If we calculate $Q_{32}(k^*_a, a)$ for $a = 0, 1, \ldots, 31$, the range is from 40,920 to 3,839,160. However, if we had just used $k' = 40$ (which is the number of odd powers in $S_k(x^8)$ for $k = 80$), the maximum $Q_{32}(40, m)$ occurs for $m \equiv 20 \pmod{32}$, and is $Q_{32}(40, 20) =$ 137,846,528,820. In contrast, $k' = 26$ is sufficient for $m \equiv 20 \pmod{32}$, and because $Q_{32}(26, 20) =$ 230,230, the more efficient version is more than 5 orders of magnitude faster. Thus the optimization involving k' to use the minimum number of odd powers necessary is critical.

In the final step, we went through every congruence class $m = 0, 1, \ldots, 2^{16} - 1 \pmod{2^{16}}$ to check whether the conditions of Theorem 3 are satisfied, and to see if there are any larger candidates for the threshold of completeness. In every case, Algorithm 13 returned (true), indicating that the conditions are met, and while checking the $m = 2466$ case, we found the largest candidate $c = T(S(x^8)) =$ 4,968,618,780,985,762. This required 22.6 days of computation time. Fortunately, we required only 13.6 GB of memory and were able to run two instances concurrently on a 32 GB laptop. We were also fortunate that the final candidate c is smaller than the value of t we found early on, so it was not necessary to restart the search through all congruence classes when we found a larger candidate c.

7 Exponents 9 to 12

Starting with exponent $n = 9$, finding thresholds of completeness requires far more computing power than we had available. We had to content ourselves with random searching of promising congruence classes mod 2^{gn} to find the largest non-summable numbers possible within our computation constraints.

For exponent $n = 9$, we did an initial exploration to get to $k = 73$, and $t \approx 1.5 \times 10^{17}$. We found that using $g = 2$ required about 110 GB, so we had to go to $g = 3$ to reduce this to about 220 MB. Thus each run with Algorithm 13 covered numbers in one congruence class mod 2^{27}. Initially, each congruence class took about 3 hours to complete. Fortunately, a new idea reduced this time considerably. For exponents $n \geq 9$, the time required to perform Algorithm 12 dominates the time required for Algorithm 5. So, the new idea is to periodically pause Algorithm 12, make a temporary copy of the memory bits $b_0, \ldots, b_\mu$, and perform Algorithm 5 on this temporary copy to see if we have already proven that all numbers in congruence class m mod 2^{27} in the range from $t+1$ to $t+(k+1)^9$ are summable so that we can quit early. This reduced the average time to check a congruence class mod 2^{27} from 3 hours to 51 minutes. However, the total run time for all congruence classes is still about 13 millennia. So, we were content to choose several thousand congruence classes at random and report the largest candidate for threshold of completeness we could find (see Table 1). A side effect of this improvement is that for $n \geq 9$ we do not need the $n = 8$ trick of using only a subset of the set of powers $S_k(x^n)$ (see Section 4.6); if a congruence class is slow because it produces many sums of powers, it will likely be possible to quit early using this new improvement. We used this improvement for all exponents 9 and larger.

We used an additional idea for the even exponents that we will illustrate using the $n = 12$ case. For $n = 12$ and $g = 3$, Algorithm 13 seeks subindexes I with $\sigma_{12}(I) \equiv m \pmod{2^{36}}$. By Theorem 11, because exponent 12 is divisible by 4, the Hamming weight of such a subindex I must be congruent to $\sigma_{12}(I)$ mod 16. So, the Hamming weight of I must be congruent to m mod 16. The number of subindexes $I < 2^k$ satisfying this Hamming weight requirement is

$$Q_{16}(\lceil k/2 \rceil, m) = \sum_{\substack{i \equiv m \pmod{16} \\ 0 \leq i \leq \lceil k/2 \rceil}} \binom{\lceil k/2 \rceil}{i}.$$

This is a minimum when $m \equiv \lfloor (k+1)/4 \rfloor - 8 \pmod{16}$. We judged these values of m to be the most promising for finding large candidates for c and t with Algorithm 13. In case this reasoning is incorrect, we tried some values of m from other congruence classes mod 16, but found that they did not give better results.

For exponents 9 to 12, we were able to check enough congruence classes m that we appeared to find parameters k and t in the correct range for Theorem 3. Table 3 shows the best parameters we found for each of these exponents as well as the estimated number of cores required to check all congruence classes and find the threshold of completeness within one year. The corresponding largest candidates we found for the threshold of completeness are in Table 1. When we check other congruence classes m, we fail to improve on these parameters

n	k	g	t	Core-Years to Complete
9	74	3	160,170,705,729,301,541	1.3×10^4
10	74	3	14,784,582,277,082,304,038	1.9×10^5
11	79	4	1,692,580,057,933,058,867,601	5.1×10^9
12	89	4	640,711,717,768,919,776,264,350	1.2×10^{12}

Table 3: Best results so far for exponents 9 to 12.

in the vast majority of cases. However, because we have checked such a small fraction of available congruence classes, the likelihood that we have found the actual threshold of completeness is very low.

8 Exponents 13 to 15

For exponents 13 to 15, every run of Algorithm 13 produced a new larger candidate c for the threshold of completeness. This means that we are very likely not in the correct range of parameter values k and t to satisfy the conditions of Theorem 3. The values of k we used for these exponents were 96, 96, and 98, respectively. For each exponent, the last run took weeks to complete, and the next run with larger parameter values would likely take even longer. Further, the lower bounds on threshold of completeness values in Table 1 for these exponents contain suspicious strings of 9s. This is because we happened to choose round numbers for the limit L, and Algorithm 13 found a non-summable value not much below L. The actual thresholds of completeness could easily be orders of magnitude larger than our best efforts so far for these exponents.

9 Exponent 16

Using Theorem 11, we know that if we are examining congruence class m mod 2^{16g} for exponent 16, the Hamming weight of subindexes must be congruent to m (mod 64). We modified Algorithm 12 to search through only those subindexes with the correct Hamming weight rather than use the pre-computed sorted table of sums of odd powers (see Section 4.5). This sped up the calculations significantly allowing us to complete the last run that produced the $n = 16$ entry in Table 1 in a few weeks despite the fact that we needed to use $k = 164$, which is significantly larger than the values we needed for smaller exponents. However, there is no reason to believe that we are particularly close to the actual threshold of completeness.

10 Conclusion

The thresholds of completeness for sets of positive nth powers were known for exponents up to 7. Our main contribution is to show that the threshold of completeness for $n = 8$ is $T(S(x^8)) = 4{,}968{,}618{,}780{,}985{,}762$. We also found lower bounds for exponents 9 to 16. Using 200,000 computer cores, our methods could find $T(S(x^9))$ in about a month and $T(S(x^{10}))$ in about a year. But the computing power required for higher exponents is impractical without significant improvements to our methods.

References

[1] R. E. Dressler and T. Parker, 12,758, *Math. Comp.* **52** (1974), 313–314.

[2] C. Fuller and R. H. Nichols, Generalized anti-Waring numbers, *J. Integer Sequences* **18** (2015), Article 15.10.5.

[3] R. L. Graham, Complete sequences of polynomial values, *Duke Math. J.* **31** (1964), 275–285.

[4] D. Kim, On the largest integer that is not a sum of distinct positive nth powers, *J. Integer Sequences* **20** (2017), Article 17.7.5.

[5] S. Lin, Computer experiments on sequences which form integral bases, in J. Leech, ed., *Computational Problems in Abstract Algebra*, Pergamon Press, 1970, pp. 365–370.

[6] M. Morse, Recurrent geodesics on a surface of negative curvature, *Trans. Amer. Math. Soc.* **22** (1921), 84–100.

[7] C. Patterson, *The Derivation of a High Speed Sieve Device*, Ph.D. thesis, University of Calgary, 1992.

[8] Š. Porubský, Richert's theorem, preprint, 2009, https://www.cs.cas.cz/portal/AlgoMath/NumberTheory/AdditiveNumberTheory/RichertTheorem.htm.

[9] E. Prouhet, Mémoire sur quelques relations entre les puissances des nombres, *C. R. Math. Acad. Sci. Paris* **33** (1851), 225.

[10] H. Richert, Über Zerfällungen in ungleiche Primzahlen, *Math. Z.* **52** (1949), 342–343.

[11] K. F. Roth and G. Szekeres, Some asymptotic formulae in the theory of partitions, *Q. J. Math.* **5** (1954), 241–259.

[12] W. Sierpiński, *Elementary Theory of Numbers*, 2nd ed., translated from Polish, A. Schinzel, ed., North-Holland and PWN-Polish Scientific Publishers, Vol. 31, 1988.

[13] N. J. A. Sloane et al., The On-Line Encyclopedia of Integer Sequences, 2023. Available at https://oeis.org.

[14] R. Sprague, Über Zerlegungen in ungleiche Quadratzahlen, *Math. Z.* **51** (1948), 289–290.

[15] R. Sprague, Über Zerlegungen in n-te Potenzen mit lauter verschiedenen Grundzahlen, *Math. Z.* **51** (1948), 466–468.

[16] A. Thue, Über unendliche Zeichenreihen, *Norske vid. Selsk. Skr. Mat. Nat. Kl.* **7** (1906), 1–22. Reprinted in T. Nagell, ed., *Selected Mathematical Papers of Axel Thue*, Universitetsforlaget, Oslo, 1977, pp. 139–158.

[17] A. Thue, Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen, *Norske vid. Selsk. Skr. Mat. Nat. Kl.* **1** (1912), 1–67. Reprinted in T. Nagell, ed., *Selected Mathematical Papers of Axel Thue*, Universitetsforlaget, Oslo, 1977, pp. 413–478.

[18] H. Warren, Jr., *Hacker's Delight*, Pearson Education Inc., Addison-Wesley, 2003.

[19] E. M. Wright, Prouhet's 1851 solution of the Tarry-Escott problem of 1910, *Amer. Math. Monthly* **66** (1959), 199–201.

2010 *Mathematics Subject Classification*: Primary 11P05.
Keywords: threshold of completeness, anti-Waring number, sum of powers.

(Concerned with sequences A001661 and A010060.)

Received January 21 2023; revised versions received February 4 2023; May 16 2023; June 1 2023. Published in *Journal of Integer Sequences*, June 6 2023.

Return to Journal of Integer Sequences home page.