# AN ALGORITHM AND ESTIMATES FOR THE ERDŐS-SELFRIDGE FUNCTION

BRIANNA SORENSON, JONATHAN P. SORENSON, AND JONATHAN WEBSTER

ABSTRACT. Let $p(n)$ denote the smallest prime divisor of the integer $n$. Define the function $g(k)$ to be the smallest integer $> k+1$ such that $p(\binom{g(k)}{k}) > k$. We present a new algorithm to compute the value of $g(k)$, and use it to both verify previous work [2, 15, 11] and compute new values of $g(k)$, with our current limit being

$$g(375) = 12\ 86399\ 96537\ 88432\ 18438\ 16804\ 13559.$$

We prove that our algorithm runs in time sublinear in $g(k)$, and under the assumption of a reasonable heuristic, its running time is

$$g(k)\exp[-c(k\log\log k)/(\log k)^2(1+o(1))] \text{ for } c>0.$$

## 1. Introduction

As stated in the abstract above, let $p(n)$ denote the smallest prime divisor of the integer $n$, and define the function $g(k)$ to be the smallest integer $> k+1$ such that $p(\binom{g(k)}{k}) > k$. So we have $g(2) = 6$ and $g(3) = g(4) = 7$.

We begin with a discussion of previous work on $g(k)$, then state our new results, and finally outline the rest of this paper.

### 1.1. Previous Work.

Paul Erdős introduced the problem of estimating the function $g(k)$ in 1969 [4]. He, along with Ecklund and Selfridge [2] showed that $g(k) > k^{1+c}$ for a small constant $c$, showed that $g(k) < e^{k(1+o(1))}$, and tabulated $g(k)$ up to $k = 40$, plus $g(42)$, $g(46)$, and $g(52)$.

Scheidler and Williams [15] described how to use Kummer's theorem to construct a sieving problem to compute $g(k)$, and they proceeded to find $g(k)$ for all $k \le 140$. Five years later, Lukes, Scheidler, and Williams [11] improved their sieve, used special-purpose hardware, and computed $g(k)$ for all $k \le 200$.

Successive analytic improvements on lower bounds of $g(k)$ have been proved by [3, 6, 10], where the strongest result known, due to Konyagin, is

$$g(k) > k^{c\log k} \text{ for } c > 0.$$

We are aware of no further results on $g(k)$ that postdate 1999.

### 1.2. Definitions and New Results.

In computing $g(k)$ for $k \le 200$, the authors of [15, 11] used Kummer's Theorem to construct a sieving problem.

**Theorem 1.1** (Kummer). *Let $k < n$ be positive integers, and let $p$ be a prime $\le k$. Let $t$ be a positive integer with $t \ge \lfloor \log_p n \rfloor$. Write*

$$k = \sum_{i=0}^{t} a_i p^i \quad \text{and} \quad n = \sum_{i=0}^{t} b_i p^i$$

*as the base-p representations of k and n respectively. Then p does not divide* $\binom{n}{k}$ *if and only if* $b_i \geq a_i$ *for* $i = 0, \ldots, t$.

For each prime $p \leq k$, this theorem gives congruences $g(k)$ must satisfy. Our approach is similar to [15, 11], but we selectively choose enough prime power moduli so that we expect $g(k)$ to be among the residues. This approach is a search for a least residue and avoids explicit sieving. We accomplish this by using the space-saving wheel which was described in [16]. This wheel data structure has been successfully used in other sieving problems [17, 18, 19] but we omit the "sieving" part that occurs after the residue is constructed. Our resulting algorithm has, so far, verified all previous computations for $g(k)$, and extended them for all $k \leq 375$.

Let $M_k := \prod_{p \leq k} p^{\lfloor \log_p k \rfloor + 1}$ and let $R_k$ denote the number of acceptable residues, under Kummer's theorem, modulo $M_k$. Then $g(k)$ is the least residue (greater than $k+1$) among the $R_k$ residues. Our *uniform distribution heuristic* (UDH) states that the $R_k$ residues are, in a sense, uniformly distributed. Under this assumption, we expect $g(k)$ to be roughly $M_k/R_k$. In fact, we define $\hat{g}(k) := M_k/R_k$. The authors of [11] studied this approximating function; it plays a central role in the analysis of our algorithm, but not in its correctness.

Assuming the UDH implies that with high probability, we have

$$\log g(k) = \log \hat{g}(k) + O(\log k).$$

Let $G(x,k)$ count the number of $n \leq x$ such that $p(\binom{n}{k}) > k$. We show unconditionally that, for $x > x_0(k)$,

$$G(x,k) = (x/\hat{g}(k))(1 + o(1)).$$

These results imply that $\hat{g}(k)$ should approximate $g(k)$ reasonably well. We then show that

$$0.53068\ldots + o(1) \quad \leq \quad \frac{\log \hat{g}(k)}{k/\log k} \quad \leq \quad 1 + o(1).$$

We prove a running time for our algorithm of

$$g(k) \exp\left[-c \frac{k \log\log k}{(\log k)^2}\right]$$

for a constant $c > 0$. We also sketch a more general argument showing our algorithm's running time is sublinear in $g(k)$, unconditionally.

1.3. **Outline.** Our paper is organized as follows. In §2 we present our algorithm, including a description of the space-saving wheel data structure. In §3 we discuss the knapsack subproblem and techniques for splitting prime rings when deciding the sieving modulus for the algorithm. In §4 we demonstrate each of the above steps to compute $g(10) = 46$. In §5 we provide some statistical evidence for the credibility of the UDH, show that $g(k)$ is roughly $\hat{g}(k)$ with high probability, and we give an easy proof of our estimate for $G(x,k)$. In §6, we show $\log \hat{g}(k)$ is proportional to $k/\log k$ and bound the running time of our algorithm. In §7, we conclude with some computational notes.

## 2. The Algorithm

The naive approach is to search through all the $R_k$ admissible residues modulo $M_k$ to find the smallest $> k+1$. However, $R_k$ is typically too large for this, making this algorithm practical only for very small $k$.

Instead, we enumerate residues that satisfy the requirements of Kummer's theorem modulo $N$, where $N$ is a divisor of $M_k$ that is larger than, but near to $g(k)$.

(1) Compute $M_k$, $R_k$, and $k\hat{g}(k) = kM_k/R_k$.
(2) Choose a divisor $N$ of $M_k$ just above our estimate $k\hat{g}(k)$ with the property that there is a minimal number of residues to check. Details of how to do this are discussed in §3.
(3) Build a *ring* data structure for each prime power dividing $N$ which is a list of admissible residues as defined by Kummer's theorem.
(4) Construct a *wheel* data structure [1] with jump tables to generate the residues modulo $N$; see [16]. A jump entry is the minimum amount to add that preserves the residue class modulo earlier rings, and jumps to an admissible residue for the current ring. [2]
(5) Rings for the remaining prime powers are also created, but not a wheel (the jumps are not needed). We refer to these rings as *filters* [2] . A residue passes the filter if, when reduced modulo the ring size, the corresponding admissible bit is set to one. The smallest residue generated from the wheel that also passes all the filters is $g(k)$.

    Any prime power ring that is part of the wheel, where that prime power fully divides $M_k$, is not needed as a filter. Or in other words, if a prime divides $N$ but not $M_k/N$, its prime power is not needed as a filter.
(6) Now that our data structures are initialized, we generate each residue modulo $N$ from the wheel to see if it passes the filters. As we go, we maintain the value of the minimum residue, so far, that passed all the filters. Once every residue from the wheel is generated, this minimum is $g(k)$.

If we run the whole algorithm and fail to find a residue that passes the filters, this means $g(k) > N$. In this case, we simply multiply our previous estimate for $g(k)$ by $k$, choose a new, larger $N$, and try again.

Note that the problem of finding a solution below a given bound to a system of pairwise coprime modular congruences is known to be NP-Complete. See [5, 12].

## 3. Prime Splitting and Knapsack

The purpose of this section is to look at how to choose $N$, a divisor of $M_k$ that is just larger than our estimate for $g(k)$. We want to choose $N$ so that the prime powers dividing $N$ give a very low filter rate, thereby giving fewer residues to enumerate, which makes the algorithm faster.

Note that selecting prime power moduli based on filter rate alone is not optimal. The size of the modulus matters as well; a smaller modulus with a higher but still good filter rate can be preferable to a large modulus with a better filter rate.

3.1. **Knapsack Problem Setup.** Let $t_p := \lfloor \log_p k \rfloor + 1$ be the number of digits required to write $k$ in base $p$, with the $a_{ip}$ representing these digits, so that $k = \sum_{i=0}^{t_p-1} a_{ip} p^i$. We have $t_p \geq 2$, and for most primes $t_p = 2$. Define $T_p$ to be

[1]Any data structure that can access residues in constant time will suffice. An anonymous referee kindly pointed out that doubly-focused enumeration[1] will work here as well. It will require more space and the early abort strategy described in Section 4 is a little harder to implement.

[2] The ordering of the rings does not matter for correctness. For speed, it is best to put the ring with the most jump entries last and put the best filters first.

the maximum exponent of $p$ so that $p^{T_p} \mid N$. This implies $0 \le T_p \le t_p$, and $N = \prod_{p \le k} p^{T_p}$.

Let $r_{ip} := p - a_{ip}$, and let $R_{xp} := \prod_{i<x} r_{ip}$. Then the number of acceptable residues modulo $p^{T_p}$ is $R_{T_p p}$. The running time of the algorithm is proportional to the number of residues modulo $N$, which, by the Chinese remainder theorem, is

$$\prod_{p \le k} R_{T_p p} \quad = \quad \prod_{p \le k} p^{T_p} \frac{R_{T_p p}}{p^{T_p}} \quad = \quad N \cdot \prod_{p \le k} \frac{R_{T_p p}}{p^{T_p}}.$$

We want to minimize the product of the filtering rates for primes included in $N$, which is equivalent to maximizing the reciprocal, which we write this way:

$$\prod_{p \le k} \frac{p^{T_p}}{R_{T_p p}} \quad = \quad \exp \sum_{p \le k} \log \frac{p^{T_p}}{R_{T_p p}}.$$

This allows us to set up a *knapsack problem*[9] for choosing prime powers to include in $N$ by setting the overall capacity of the knapsack to $\log N$, and the size and value of prime powers are set as follows:

$$\begin{aligned} \text{size}(p^T) \quad &:= \quad \log p^T = T \log p \\ \text{value}(p^T) \quad &:= \quad \log(\text{modulus}/\#\text{ residues}) = \log(p^T/R_T) = T \log p - \log R_T \end{aligned}$$

The question, then, is how to set $T$ for each prime $p$ to give a good selection of items to include in the knapsack. Also, we must ensure that the same prime $p$ is not chosen more than once, with different $T$ values, for inclusion in the knapsack.

Asymptotically, we show in §6 that the expected size of $\log N$ is roughly $k/\log k$, so that only roughly $k/(\log k)^2$ primes are needed in $N$, allowing an average filter rate of about $1/\log k$ for each prime, and that $T_p$ can be set to 1 for the primes included in $N$.

3.2. **Prime Splitting.** In practice, we can often get better results by including prime powers. So our approach is, for each prime $p \le k$, to compute an optimal value for $T$ based on filter rate, and then use a greedy algorithm to fill our knapsack. We call computing this value for $T$ *splitting* the prime power, and label this split point $s_p$. We then allow for up to three possible choices in the knapsack for each prime $p$: set $T = 0$ (that is, omit $p$ from $N$ entirely), use $T = s_p$ (use the optimal split point), or use $T = t_p$, the maximum (note that $s_p = t_p$ is possible).

Maximizing the value-to-size ratio, we get

$$\frac{\text{value}}{\text{size}} \quad = \quad \frac{T \log p - \log R_{Tp}}{T \log p} \quad = \quad 1 - \frac{\log R_{Tp}}{T \log p}.$$

So, in time linear in $t_p$, we can try all possible $T$ values and quickly find the optimum, $s_p$. Since 1 and $\log p$ do not change, it suffices to compute $(1/T) \log R_{Tp}$ for each $T$ to find the optimum.

3.3. **The Greedy Knapsack Algorithm.** After splitting, we have a list of candidate prime powers to include in $N$. We sort the list based on value-to-size ratio, and choose enough to include in $N$ based on the value of $\hat{g}(k)$. In practice, this simple and fast algorithm to construct $N$ worked very well.

3.4. **A Dynamic Programming Approach.** An anonymous referee pointed out an elegant way to find $N$.

Start with $(N = 1, R = 1)$, where $N$ is the modulus, and $R$ the number of admissible residues. For each prime power $p^t$ appearing in $M_k$, and for each $(N, R)$ value found so far, form new values $(N \cdot p^i, R \cdot R_{ip})$ for $0 \le i \le t$, where $R_{ip}$ is the number of admissible residues modulo $p^i$. Sort the new $(N, R)$ values by increasing value of $R$. For each $(N, R)$, $(N', R')$ with $R < R'$, discard $(N', R')$ if $N' < N$, since $(N, R)$ is always better. Also discard values $(N', R')$ if $N' \ge N \ge k\hat{g}(k)$.

This clever algorithm will create an optimal solution for $N$, but at first glance appears to have a running time that is superpolynomial in $k$. We have not yet implemented this, but it might be worth the effort.

## 4. Example for $g(10)$

As an example computation, we present each of the steps described above to compute $g(10) = 42$.

We write $10 = 1010_2 = 101_3 = 20_5 = 13_7$. Kummer's Theorem then says that $g(10) \equiv 1010_2, 1011_2, 1110_2, 1111_2 \bmod 16$. Similarly, there are 12 residues modulo $3^3$, 15 residues modulo $5^2$, and 24 residues modulo $7^2$. In total, there are $R_{10} = 4{\cdot}12{\cdot}15{\cdot}24 = 17280$ admissible residues modulo $M_{10} = 16{\cdot}27{\cdot}25{\cdot}49 = 529200$. We compute $10 \cdot \hat{g}(10) = 306.25$ for use in our knapsack problem.

Considering the powers of 2 first, we compute $r_{12} = 2$, $r_{22} = 1$, $r_{32} = 2$, and $r_{42} = 1$. This gives $R_{12} = 2$, $R_{22} = 2$, $R_{32} = 4$, and $R_{42} = 4$. We get value-to-size ratios of 0, 1/2, 1/3, and 1/2. This implies $s_2 = 2$ or 4. In practice, we normally use the largest value for $s_p$ when several values give the same ratio, since it implies a better filter rate.

For $p = 3$, we have $k = 101_3$. We have $r_{13} = 2$, $r_{23} = 3$, and $r_{33} = 2$. This gives $R_{13} = 2$, $R_{23} = 6$, and $R_{33} = 12$. The successive $(1/T) \log R$ values are $\log 2$, $(1/2) \log 6$, and $(1/3) \log 12$. Of these, $\log 2$ is the smallest, giving $s_3 = 1$. In a similar fashion, we obtain $s_5 = 2$ and $s_7 = 1$.

We then construct the following table (using the natural logarithm):

| $p$ | $T$ | *value* | *size* | *ratio* |
|---|---|---|---|---|
| 2 | 4 | $\log(2^4/4)$ | $\log(2^4)$ | 0.5 |
| 3 | 1 | $\log(3/2)$ | $\log 3$ | 0.4009... |
| 3 | 3 | $\log(3^3/12)$ | $\log(3^3)$ | 0.246... |
| 5 | 2 | $\log(5^2/20)$ | $\log(5^2)$ | 0.069... |
| 7 | 1 | $\log(7/4)$ | $\log 7$ | 0.287... |
| 7 | 2 | $\log(7^2/24)$ | $\log(7^2)$ | 0.183... |

We use a greedy algorithm to choose items to include in our knapsack of size $\log 306$. We first choose $2^4 = 16$, leaving $306/16 \approx 20$ "room" in our knapsack; then 3 is chosen next. This leaves about $20/3 \approx 7$ room. The choice of 7 fills all remaining room, and gives $N = 2^4 \cdot 3 \cdot 7$.

Using $N$ we set up the space saving wheel with rings that encode $g(10) \equiv 10, 11, 14, 15 \pmod{16}$, $g(10) \equiv 1, 2 \pmod{3}$, and $g(10) \equiv 3, 4, 5, 6 \pmod{7}$. If $N$ is large enough, we expect $g(10)$ to be among these 32 residues.

The jump tables are:

Ring 16:

| residue | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| admissible | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| jump | +10 | +9 | +8 | +7 | +6 | +5 | +4 | +3 | +2 | +1 | +1 | +3 | +2 | +1 | +1 | +11 |

Ring 3:

| residue | 0 | 1 | 2 |
|---|---|---|---|
| admissible | 0 | 1 | 1 |
| jump | +16 | +16 | +32 |

Ring 7:

| residue | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| admissible | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| jump | +48 | +96 | +144 | +192 | +48 | +48 | +48 |

We would also build filters for the prime power congruences not used in the jump tables, but omit their explicit construction for the sake of brevity.

The smallest possible starting point is $k+2$, or 12 in our example. Since 12 is not admissible modulo 16, we apply the jump (+2) to get 14. We pass up to the next ring. We find $14 \bmod 3 \equiv 2$ is admissible. We pass to the next ring. Since $14 \bmod 7 \equiv 0$ is not admissible, we jump (+48) to get 62. There are 4 total residues in the 7 ring, so we also generate $62+48 = 110$, $110+48 = 158$, and $158+48 = 206$. All residues produced by the 7 ring are filtered:

$$62 \bmod 27 \equiv 8 = 22_3, \text{ fail} \qquad 110 \bmod 27 \equiv 2_3, \text{ fail}$$
$$158 \bmod 25 \equiv 8 = 13_5, \text{ fail} \qquad 206 \bmod 25 \equiv 6 = 11_5, \text{ fail}$$

We then backtrack to ring 3 at 14, and generate $14+32 = 46$. We pass to ring 7. The initial value in this ring, $46 \bmod 7 \equiv 4$, is already admissible and is generated first. We find that 46 passes all filters. We record this value as a candidate for $g(10)$ and continue the computation to see if a smaller value exists. Since, $g(10) = 46$, no such value will be found. Note that nothing larger than $N$ can be generated.

After 4 residues in the 7 ring, we drop down to the 3 ring, where we have already done 2 residues, so we drop back to the 16 ring. At the 16 ring, we generate the next residue $14 + 1 = 15$, which is passed up to the 3 ring.

This implies that, at each ring, we need to keep track of the next residue to generate, and how many have been generated so far so that we know when to back up to a previous ring.

And so it goes. The amortized cost is a constant number of arithmetic operations per residue generated by the outermost ring where they are filtered. If we apply the filters in decreasing order of filter rate, on average, a residue is only tested against a constant number of filters, and so again, the cost is a constant number of arithmetic operations per residue modulo $N$.

By keeping track of the minimum residue that passes the filters, we do not have to generate any residues larger than this minimum. In our example, once 46 passes the filters, we don't even generate the rest of ring 7. This optimization can make a big difference in practice.

## 5. Uniform Distribution Heuristic

The *Uniform Distribution Heuristic* (UDH) states that the admissible residues modulo $M_k$ behave as if they are chosen at random from a uniform distribution over the interval $[1, M_k - 1]$. It is not entirely dissimilar to Cramér's random model, the heuristic that integers $\leq x$ are prime with probability $1/\log x$, and our intention is that these two models be treated similarly, in that we know they are not, strictly speaking, true, yet seem to have good predictive behavior under the right circumstances.

With the help of Rasitha Jayasekare, a statistician at Butler University, we ran statistical tests on the residues for $5 \le k \le 15$. For each $k$ we generated all $R_k$ admissible residues and used the the Anderson-Darling and Kolmogorov-Smirnov tests to measure uniformity. Both tests confirm with a high probability that the data comes from a uniform distribution.

**Theorem 5.1.** *The UDH implies that, with probability* $1 - o(1)$*, we have*

$$\hat{g}(k)/k \le g(k) \le k\hat{g}(k).$$

*Proof.* WLOG, we ignore residues $\le k+1$ because $k$ is asymptotically negligible compared to $M_k$ and $R_k$.

We have

$$\begin{aligned} Pr(g(k) \le x) &= 1 - Pr(\text{ all residues are } > x) \\ &= 1 - \left(\frac{M_k - x}{M_k}\right)^{R_k} = 1 - \left(1 - \frac{x}{M_k}\right)^{R_k} \end{aligned}$$

For an upper bound, set $x = (kM_k)/R_k$, to obtain

$$Pr(g(k) \le (kM_k)/R_k) = 1 - \left(1 - \frac{k}{R_k}\right)^{R_k} \sim 1 - e^{-k} = 1 - o(1)$$

for large $R_k$ (and $R_k$ does get quite large).

For a lower bound, set $x = M_k/(kR_k)$ to obtain

$$Pr(g(k) \le M_k/(kR_k)) = 1 - \left(1 - \frac{1}{kR_k}\right)^{R_k} \sim 1 - e^{-1/k} = o(1).$$

This completes the proof. □

So we have that, with high probability,

$$\log g(k) = \log \hat{g}(k) + O(\log k)$$

if we assume the uniform distribution heuristic. This has worked well in practice; the inequality in Theorem 5.1 is satisfied by all computed $g(k)$ (excepting $k = 99$).

Recall that $G(x,k)$ counts the integers $n \le x$ such that $p(\binom{n}{k}) > k$. We conclude this section with the following.

**Theorem 5.2.** *If $x$ is sufficiently large, then* $G(x,k) = (x/\hat{g}(k))(1 + o(1))$.

*Proof.* Write $x = q \cdot M_k + r$ using the division algorithm, with integers $q, r > 0$ and $r < M_k$. A contiguous interval of length $M_k$ will have exactly $R_k$ admissible residues, so $G(qM_k, k) = qR_k$. The remaining interval of length $r$ has at most $R_k$ residues, so $G(x,k) = G(qM_k,k) + O(R_k) = qR_k + O(R_k)$ but $q = \lfloor x/M_k \rfloor$, so

$$G(x,k) = \lfloor x/M_k \rfloor R_k + O(R_k) = (x/\hat{g}(k))(1 + o(1)).$$

□

## 6. Analysis

The running time of our algorithm is linear in the number of residues modulo $N$. Since we choose $N$ based on $\hat{g}(k)$, we need to estimate $\hat{g}(k)$.

**Theorem 6.1.**

$$0.53068\ldots + o(1) \le \frac{\hat{g}(k)}{k/\log k} \le 1 + o(1).$$

Applying the definitions for $M_k$ and $R_k$ above, we have

$$\begin{aligned}
\hat{g}(k) = \frac{M_k}{R_k} &= \frac{\prod_{p\le k} p^{\lfloor \log_p k\rfloor+1}}{\prod_{p\le k}\prod_{i=0}^{\lfloor \log_p k\rfloor}(p-a_{ip})} = \prod_{p\le k}\prod_{i=0}^{\lfloor \log_p k\rfloor}\frac{p}{p-a_{ip}} \\
&= \prod_{p\le\sqrt{k}}\prod_{i=0}^{\lfloor \log_p k\rfloor}\frac{p}{p-a_{ip}} \cdot \prod_{\sqrt{k}<p\le k}\prod_{i=0}^{\lfloor \log_p k\rfloor}\frac{p}{p-a_{ip}} \\
&= \prod_{p\le\sqrt{k}}\prod_{i=0}^{\lfloor \log_p k\rfloor}\frac{p}{p-a_{ip}} \cdot \prod_{\sqrt{k}<p\le k}\frac{p}{p-a_{1p}}\frac{p}{p-a_{0p}}.
\end{aligned}$$

Here we observed that $\lfloor \log_p k\rfloor + 1 = 2$ when $p > \sqrt{k}$.

We will show that the product on the factor involving $a_{0p}$ is exponential in $k/\log k$, and is therefore significant; and the other two factors, the product on primes up to $\sqrt{k}$, and the factor with $a_{1p}$, are both only exponential in $\sqrt{k}$.

We bound the first product, on $p \le \sqrt{k}$, with the following lemma.

**Lemma 6.2.**

$$\prod_{p\le\sqrt{k}}\prod_{i=0}^{\lfloor \log_p k\rfloor}\frac{p}{p-a_{ip}} \quad\ll\quad e^{3\sqrt{k}(1+o(1))}.$$

*Proof.* We note that $a_{ip} \le p-1$, giving

$$\prod_{p\le\sqrt{k}}\prod_{i=0}^{\lfloor \log_p k\rfloor}\frac{p}{p-a_{ip}} \quad\le\quad \prod_{p\le\sqrt{k}} p^{\lfloor \log_p k\rfloor+1} \quad\le\quad \prod_{p\le\sqrt{k}} p^{3\lfloor \log_p \sqrt{k}\rfloor}.$$

From [7, Ch. 22] we have the bound

$$\sum_{p\le x}\lfloor \log_p x\rfloor \log p = x(1+o(1)). \tag{6.1}$$

Exponentiating and substituting $\sqrt{k}$ for $x$ gives the desired result. □

Next, we show that the product involving $a_{1p}$ is small.

**Lemma 6.3.**

$$\prod_{\sqrt{k}<p\le k}\frac{p}{p-a_{1p}} \quad\ll\quad 2^{\sqrt{k}}.$$

*Proof.* Observe that for any prime $p$ with $\sqrt{k} < p \le k$, if $a_{1p} = a$, then $k/(a+1) < p \le k/a$. We have

$$\begin{aligned}
\prod_{\sqrt{k}<p\le k} \frac{p}{p-a_{1p}} &= \prod_{a=1}^{\lfloor\sqrt{k}\rfloor} \prod_{k/(a+1)<p\le k/a} \frac{p}{p-a} = \prod_{a=1}^{\lfloor\sqrt{k}\rfloor} \prod_{k/(a+1)<p\le k/a} \left(1-\frac{a}{p}\right)^{-1} \\
&= \prod_{a=1}^{\lfloor\sqrt{k}\rfloor} \frac{\prod_{a<p\le k/a}\left(1-\frac{a}{p}\right)^{-1}}{\prod_{a<p\le k/(a+1)}\left(1-\frac{a}{p}\right)^{-1}} \\
&= \prod_{a=1}^{\lfloor\sqrt{k}\rfloor} \frac{(c(a)\log(k/a))^a(1+o(1))}{(c(a)\log(k/(a+1)))^a(1+o(1))} \\
&= (1+o(1))\frac{\log k}{\log(k/2)}\cdot\left(\frac{\log(k/2)}{\log(k/3)}\right)^2\cdot\left(\frac{\log(k/3)}{\log(k/4)}\right)^3\cdots\left(\frac{\log(\frac{k}{\lfloor\sqrt{k}\rfloor})}{\log(\frac{k}{\lfloor\sqrt{k}\rfloor+1})}\right)^{\lfloor\sqrt{k}\rfloor} \\
&= (1+o(1))\frac{\log k}{\log\sqrt{k}}\cdot\frac{\log(k/2)}{\log\sqrt{k}}\cdot\frac{\log(k/3)}{\log\sqrt{k}}\cdots\frac{\log(k/\lfloor\sqrt{k}\rfloor)}{\log\sqrt{k}} \\
&\ll 2^{\sqrt{k}}.
\end{aligned}$$

This used the following variant of Mertens's theorem, which holds for $b > 0$, where $c(b)$ is a constant that depends only on $b$:

$$\prod_{b<p\le x}\left(1-\frac{b}{p}\right) = \left(\frac{c(b)}{\log x}\right)^b(1+o(1)). \tag{6.2}$$

This is readily proved following the arguments in Hardy and Wright [7, §22.7]. □

We now have

$$\log \hat{g}(k) = \log\left(\prod_{\sqrt{k}<p<k}\frac{p}{p-a_{0p}}\right) + O(\sqrt{k}).$$

The following lemma wraps up the proof of our theorem.

**Lemma 6.4.**

$$0.53068\ldots\cdot\frac{k}{\log k}(1+o(1)) \le \log\left(\prod_{\sqrt{k}<p\le k}\frac{p}{p-a_{0p}}\right) \le \frac{k}{\log k}(1+o(1)).$$

*Proof.* Fix $a_{1p} = a$. Then $k/(a+1) < p \le k/a$, and $a_{0p} = k \bmod p = k - ap$ and $p - a_{0p} = p - (k-ap) = (a+1)p - k$. We have

$$\begin{aligned}
\log\left(\prod_{\sqrt{k}<p\le k}\frac{p}{p-a_{0p}}\right) &= \log\left(\prod_{a=1}^{\sqrt{k}}\prod_{k/(a+1)<p\le k/a}\frac{p}{(a+1)p-k}\right) \\
&= \sum_{a=1}^{\sqrt{k}}\sum_{k/(a+1)<p\le k/a}\left(\log p - \log((a+1)p-k)\right).
\end{aligned}$$

We split this sum into three pieces to start with:

(1) The outer sum for $(\log k)^2 \le a \le \sqrt{k}$, and we show it is $o(k/\log k)$.
(2) The $\log p$ term only, for $a < (\log k)^2$, and show it is $k + o(k/\log k)$.
(3) The $-\log((a+1)p-k)$ term, again for $a < (\log k)^2$, and show it is $-k + O(k/\log k)$.

For (1), we have

$$\begin{aligned}\sum_{a=(\log k)^2}^{\sqrt{k}} \sum_{k/(a+1)<p\le k/a} (\log p - \log((a+1)p-k)) &\le \sum_{a=(\log k)^2}^{\sqrt{k}} \sum_{k/(a+1)<p\le k/a} \log p \\ &\le \sum_{\sqrt{k}<p\le k/(\log k)^2} \log p\end{aligned}$$

which is $O(k/(\log k)^2)$ using $\sum_{p<x} \log p = x + o(x/\log x)$. For (2), we have

$$\sum_{a=1}^{(\log k)^2} \sum_{k/(a+1)<p\le k/a} \log p = \sum_{k/(\log k)^2<p\le k} \log p$$

which is $k + o(k/\log k)$. For (3), we have

$$-\sum_{a=1}^{(\log k)^2} \sum_{k/(a+1)<p\le k/a} \log((a+1)p-k). \tag{6.3}$$

Rewriting the inner sum as an integral, using a strong version of the prime number theorem, we get

$$\begin{aligned}&-\sum_{k/(a+1)<p\le k/a} \log((a+1)p-k) \\ &= -\int_{k/(a+1)}^{k/a} \frac{\log((a+1)t-k)}{\log t} dt + o(k/(\log k)^3) \\ &= -\frac{1}{\log(k/(a+\alpha))} \int_{k/(a+1)}^{k/a} \log((a+1)t-k)dt + o(k/(\log k)^3).\end{aligned}$$

Here $\alpha$ is between 0 and 1, determined implicitly by the mean value theorem. The precise value of $\alpha$ may depend on both $k$ and $a$. We will use either $\alpha = 0$ or $\alpha = 1$, depending on whether we want an upper or lower bound, respectively.

Using substitution, we can readily show that

$$\int_{k/(a+1)}^{k/a} \log((a+1)t-k)dt = \frac{k(\log(k/a)-1)}{a(a+1)}.$$

We have for (3), then,

$$\begin{aligned}&= \sum_{a=1}^{(\log k)^2} \left( -\frac{k(\log(k/a)-1)}{a(a+1)\log(k/(a+\alpha))} \right) \\ &= -k + \frac{k}{\log k} \cdot \sum_{a=1}^{(\log k)^2} \frac{1-\log\left(1+\frac{\alpha}{a}\right)}{a(a+1)} \cdot \left(1 + O\left(\frac{\log\log k}{\log k}\right)\right).\end{aligned}$$

The last step requires a bit of algebra, and the observation that $1/(u-v) = 1/u + v/(u(u-v))$.

To obtain the upper bound, set $\alpha = 0$, and note that $\sum 1/(a(a+1))$ converges to 1. To obtain the lower bound, set $\alpha = 1$, and note that $\sum(1-\log(1+1/a))/(a(a+1))$ converges to a constant near $0.53068\ldots$. □

**Algorithm Running Time.**

**Theorem 6.5.** *If the UDH is true, then with probability* $1 - o(1)$, *our algorithm has a running time bounded by*

$$g(k) \cdot \exp\left[\frac{-ck \log\log k}{(\log k)^2}(1 + o(1))\right]$$

*where* $c > 2$ *is constant.*

*Proof.* Without loss of generality, we assume that $g(k) \le N < k \cdot g(k)$, as we can guess a smaller $N$, run the algorithm, and if it fails to find $g(k)$, include another prime $p$ with $k/2 < p < k$ in $N$, and repeat. Since $N$ at least doubles each time we do this, the cost of running the algorithm on all $N < g(k)$, and failing, is bounded by a factor of $\log g(k)$ times the cost of the final run with a value of $N > g(k)$ that succeeds. We absorb this multiplicative factor of $\log g(k)$ in the $o(1)$ error term in the exponent of the running time bound above as $\log g(k) = \Theta(k/\log k)$ with high probability. In particular, this gives us $\log N = (1 + o(1)) \log g(k)$ with high probability.

For the purposes of this proof, we choose $N$ to be a product of some primes between $k/2$ and $k$. This is conservative, as the choice of primes or prime powers for inclusion in $N$, using the methods discussed earlier, will result in a faster algorithm in practice. So we have

$$\prod_{p|N} p = N \approx g(k)$$

and thus

$$\sum_{p|N} \log p = \log N \sim \log g(k) \ll k/\log k.$$

Since $\sum_{k/2<p\le k} \log p = (k/2)(1 + o(1))$, we have more primes in this range than we need for $N$ by a factor of roughly $(1/2)\log k$. Thus, we can choose the best $k/(\log k)^2$ primes (roughly) below $k$ of the $k/\log k$ that are available. As a result, we expect to get a filtering factor of $1/\log k$ for the primes we choose. Indeed, if we choose all primes $p$ with $k/2 < p < k/2 + c_1 k/\log k$, with $c_1 > 0$ an appropriate constant we fix later, this is the case.

Let's check that this gives us a good value for $N$. We have

$$\begin{aligned} \log N &= \sum_{k/2<p<k/2+c_1 k/\log k} \log p \\ &= \frac{c_1 k}{(\log k)^2} \log(k/2)(1 + o(1)) \quad = \quad \frac{c_1 k}{\log k}(1 + o(1)), \end{aligned}$$

which is larger than $\log g(k)$ with high probability if we choose $c_1 > 2$. (See [13, (2.29)].)

Now we address the filter rate, and hence the running time. For each such prime $p$, $k+\frac{2c_1k}{\log k} > 2p > k$, which implies $k-p > p - \frac{2c_1k}{\log k}$ so that

$$\begin{aligned} a_{0p} &= k \bmod p &&= k-p \\ &> p-\frac{2c_1k}{\log k} &&> p-\frac{4c_1p}{\log k} &&= p\left(1-\frac{4c_1}{\log k}\right). \end{aligned}$$

Our running time, then, is proportional to the number of acceptible residues modulo $N$, which is

$$\begin{aligned} \prod_{k/2<p<k/2+c_1k/\log k} (p-a_{0p}) &= \prod_p \left(p-p\left(1-\frac{4c_1}{\log k}\right)\right) \\ &= \prod_p p\cdot\frac{4c_1}{\log k} \quad = \quad N\prod_p \frac{4c_1}{\log k} \\ &\le kg(k)\left(\frac{4c_1}{\log k}\right)^{c_1k/(\log k)^2(1+o(1))} \\ &= g(k)\exp\left[-c_1\frac{k\log\log k}{(\log k)^2}(1+o(1))\right]. \end{aligned}$$

□

The UDH is stronger than what we need to prove a sublinear running time. The central issue is finding enough primes $p$ with $k/2 < p \le k/2+\Delta$ such that the product of these primes is roughly $g(k)$. If the number of primes in this interval is $\Delta/\log k$, then we can set $\Delta \approx \log g(k)$. Pushing this through our argument above, we obtain a running time of the form

$$g(k)\cdot\exp\left[\frac{-c\Delta}{\log k}\log\left(\frac{4\Delta}{k}\right)(1+o(1))\right]$$

where $c>0$ is constant. Observe that plugging in $\log g(k) \approx k/\log k$ gives our theorem, but this form is valid so long as we can find enough primes. In fact, if $\log g(k) \gg k^{\theta}$, with $7/12 < \theta \le 1$, we can use a result due to Heath-Brown [8] on primes in short intervals to guarantee this is true.

If $g(k)$ is smaller than this, we would choose $\Delta = (\log g(k)/\log k)E(k)$, where $E(k)$ is the error term for the prime number theorem for $\pi(k)$, to give us the needed $\log g(k)/\log k$ primes above $k/2$. (If we assumed the Riemann Hypothesis, this would let us use a smaller $E(k)$ term.) Pushing this through, we obtain a weaker, but still sublinear, running time.

We were also able to show that

$$\limsup_{k\to\infty}\frac{\hat{g}(k+1)}{\hat{g}(k)} = \infty.$$

We omit the proof due to a lack of space, but the interesting case is when $k+1$ is prime. It is conjectured that the same holds true for $g(k)$ itself, but that remains an open problem [2].

## 7. Computations

We conclude with a brief discussion of the timing results. More detailed timing results or a copy of the source code would be provided upon request made to the second author.

7.1. **Timing Results.** We implemented our algorithm from §2 in C++. We started with a sequential program, which we used to compute $g(k)$ for all $k \leq 272$, thereby verifying all previous computations along the way [2, 14, 11]. None of these smaller $k$ values took more than a couple hours on a standard desktop computer.

We then parallelized our algorithm, using MPI, by having each core generate a share of the residues. However, if a particular core found a new, smaller residue that passed all filters, that new upper bound would not be communicated to all the other cores for some time. This resulted in a fair amount of wasted work. On the other hand, too-frequent inter-core communication would also slow down the computation, since finding new upper bounds is a rare event. We found that our computation distributed over 192 cores only performed about 40-50 times faster than the single-core version.

Our parallel code took anywhere from under an hour to over 1300 hours to compute each $g(k)$ value. The timing results, in hours of wall time, are shown in Figure 1. Here the $y$-axis on the left is in hours, and the $y$-axis on the right is used for $g(k)$ values, which are plotted on the same graph for comparison. In total, the cluster was exclusively computing $g(k)$ values for about 9 months. The cluster is composed of Intel Xeon E5-2630 v2 processors, with 15MB cache, running at 2.3 GHz. Our algorithm uses very little memory, and so RAM is not an issue.

7.2. **Verification is Faster.** It is easy to verify that our claimed $g(k)$ values all satisfy Kummer's Theorem and are new $\hat{g}(k)$. However, we know of no way to independently verify our computations except by repeating the search. Knowing a small admissible candidate gives two significant practical advantages in our algorithm. First, you can work with a modulus $N$ just larger than the candidate $g(k)$ value, which is usually smaller than the suggested $k\hat{g}(k)$ value. Second, you can input the claimed $g(k)$ value as the starting upper bound for residues. Take the computation of $g(225)$ as an example. The initial search worked modulo $N = 1012\ 44299\ 87665\ 22178\ 24000$ and went through at most 64 66521 60000 residues. The candidate for $g(225)$ was updated three times and the computation took about 26 minutes. A verification computation was done working modulo $N = 2\ 95172\ 88593\ 77615\ 68000$, had at most 1 19750 40000 residues to check, and $g(225)$ was an input for the initial upper bound. This second computation completed in just 24 seconds. We note that a parallel version of a verification computation can also avoid some of the communication overhead.

**Acknowledgments.** The first author was supported in part by the Butler Summer Institute, by the Honors program, and by the Mathematics Research Camp at Butler University. The second and third authors were supported in part by a grant from the Holcomb Awards Committee at Butler University.

Special thanks to Rasitha Jayasekare, our friendly neighborhood statistician, for helping us with uniform distribution statistical tests. Also thanks to Michael Filaseta for his help with references.

Finally, thanks to Frank Levinson, who generously supports Butler University's computing research infrastructure.

## References

[1] Daniel J. Bernstein. Doubly focused enumeration of locally square polynomial values. In *High primes and misdemeanours: lectures in honour of the 60th birthday of Hugh Cowie Williams*, volume 41 of *Fields Inst. Commun.*, pages 69–76. Amer. Math. Soc., Providence, RI, 2004.

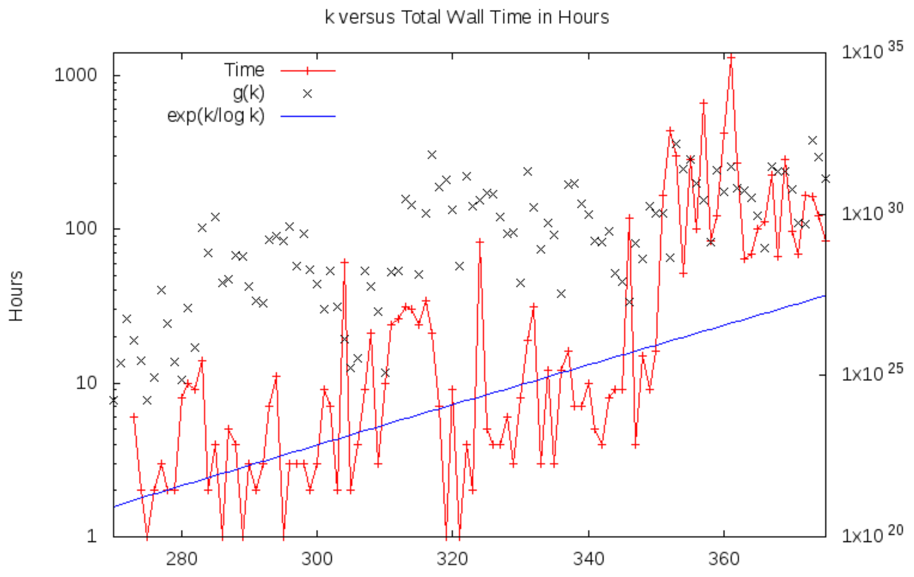

FIGURE 1. Running Time (wall time) in Hours

[2] E. F. Ecklund, Jr., P. Erdös, and J. L. Selfridge. A new function associated with the prime factors of $\binom{n}{k}$. *Math. Comp.*, 28:647–649, 1974.
[3] P. Erdős, C. B. Lacampagne, and J. L. Selfridge. Estimates of the least prime factor of a binomial coefficient. *Math. Comp.*, 61(203):215–224, 1993.
[4] Paul Erdős. Some problems in number theory. In A.O.L. Atkin and B.J. Birch, editors, *Computers in Number Theory*, pages 405–414. Academic Press, 1971.
[5] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.
[6] Andrew Granville and Olivier Ramaré. Explicit bounds on exponential sums and the scarcity of squarefree binomial coefficients. *Mathematika*, 43(1):73–107, 1996.
[7] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers.* Oxford University Press, 5th edition, 1979.
[8] D.R. Heath-Brown. The number of primes in a short interval. *Journal für die reine und angewandte Mathematik*, 389:22–63, 1988.
[9] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems.* Springer Berlin Heidelberg, 2013.
[10] S. V. Konyagin. Estimates of the least prime factor of a binomial coefficient. *Mathematika*, 46(1):4155, 1999.
[11] Richard F. Lukes, Renate Scheidler, and Hugh C. Williams. Further tabulation of the Erdős-Selfridge function. *Math. Comp.*, 66(220):1709–1717, 1997.
[12] Kenneth L. Manders and Leonard Adleman. NP-complete decision problems for binary quadratics. *Journal of Computer and System Sciences*, 16(2):168 – 184, 1978.
[13] J. B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962.
[14] R. Scheidler and H. C. Williams. A public-key cryptosystem utilizing cyclotomic fields. Technical Report 15/92, University of Manitoba, Department of Computer Science, November 1992.
[15] Renate Scheidler and Hugh C. Williams. A method of tabulating the number-theoretic function $g(k)$. *Math. Comp.*, 59(199):251–257, 1992.
[16] Jonathan P. Sorenson. The pseudosquares prime sieve. In Florian Hess, Sebastian Pauli, and Michael Pohst, editors, *Proceedings of the 7th International Symposium on Algorithmic*

*Number Theory (ANTS-VII)*, pages 193–207, Berlin, Germany, July 2006. Springer. LNCS 4076, ISBN 3-540-36075-1.

[17] Jonathan P. Sorenson. Sieving for pseudosquares and pseudocubes in parallel using doubly-focused enumeration and wheel datastructures. In Guillaume Hanrot, Francois Morain, and Emmanuel Thomé, editors, *Proceedings of the 9th International Symposium on Algorithmic Number Theory (ANTS-IX)*, pages 331–339, Nancy, France, July 2010. Springer. LNCS 6197, ISBN 978-3-642-14517-9.

[18] Jonathan P. Sorenson and Jonathan Webster. Strong pseudoprimes to twelve prime bases. *Math. Comp.*, 86(304):985–1003, 2017.

[19] Jonathan P. Sorenson and Jonathan Webster. Two algorithms to find primes in patterns. arXiv:1807.08777, 2018.

Butler University, Indianapolis, IN 46208, USA
*E-mail address*: bsorenso@butler.edu

Butler University, Indianapolis, IN 46208, USA
*E-mail address*: sorenson@butler.edu
*URL*: blue.butler.edu/∼jsorenso

Butler University, Indianapolis, IN 46208, USA
*E-mail address*: jewebste@butler.edu