ALGORITHMS FOR THE MULTIPLICATION TABLE PROBLEM

RICHARD BRENT, CARL POMERANCE, DAVID PURDUM, AND JONATHAN WEBSTER

ABSTRACT. Let $M(n)$ denote the number of distinct entries in the $n \times n$ multiplication table. The function $M(n)$ has been studied by Erdős, Tenenbaum, Ford, and others, but the asymptotic behaviour of $M(n)$ as $n \to \infty$ is not known precisely. Thus, there is some interest in algorithms for computing $M(n)$ either exactly or approximately. We compare several algorithms for computing $M(n)$ exactly, and give a new algorithm that has a subquadratic running time. We also present two Monte Carlo algorithms for approximate computation of $M(n)$. We give the results of exact computations for values of n up to 2^{30}, and of Monte Carlo computations for n up to $2^{100,000,000}$, and compare our experimental results with Ford's order-of-magnitude result.

1. Introduction

Although a multiplication table is understood by a typical student in elementary school, there remains much that we do not know about such tables. In 1955, Erdős studied the problem of counting the number $M(n)$ of distinct products in an $n \times n$ multiplication table. That is, $M(n) := |\{ij : 1 \le i, j \le n\}|$. In [12], Erdős showed that $M(n) = o(n^2)$. Five years later, in [13], he obtained

$$M(n) = \frac{n^2}{(\log n)^{c+o(1)}} \text{ as } n \to \infty, \tag{1}$$

where (here and below) $c = 1 - (1 + \log\log 2)/\log 2 \approx 0.086071$. In 2008, Ford [15, 16] gave the correct order of magnitude

$$M(n) = \Theta(n^2/\Phi(n)), \tag{2}$$

where

$$\Phi(n) := (\log n)^c (\log\log n)^{3/2} \tag{3}$$

is a slowly-growing function.[1]

Note that (2) is not a true asymptotic formula, as $M(n)/(n^2/\Phi(n))$ might or might not tend to a limit as $n \to \infty$. The computation of $M(n)$ for various large n could suggest (though not prove) the true behaviour of $M(n)$ as $n \to \infty$. The history of such computations goes back to Brent and Kung [8], who considered how much area and time are needed to perform an n-bit binary multiplication on a VLSI chip. For this, they needed a lower bound on $M(2^n - 1)$. In 1981 they computed[2]

(Brent) Australian National, University, Canberra, ACT 2600, Australia
(Pomerance) Mathematics Department, Dartmouth College, Hanover, NH 03755
(Purdum, Webster) Butler University, Indianapolis, IN 46208

2000 *Mathematics Subject Classification.* 11A25, 11N37, 11Y16, 11Y70, 65C05, 68Q25 .

[1] In (2), the notation $f = \Theta(g)$ means $f = O(g)$ and $g = O(f)$, i.e. there are positive constants A and B such that $Ag(n) \le f(n) \le Bg(n)$ for all sufficiently large n.

[2] Brent and Kung actually computed $M(2^n - 1) + 1 = |\{ij : 0 \le i, j < 2^n\}|$. For consistency in the exposition, we translate their results to the definition of $M(n)$ stated above.

$M(2^n-1)$ for $1 \le n \le 17$. In unpublished work dating from 2012, the first two authors revisited the problem, extending the computation through $n = 25$, and exploring Monte Carlo estimates for larger n. Some years later, the fourth author discovered several new algorithms for the exact computation of $M(n)$. In this paper we present both exact and Monte Carlo algorithms.

It is useful to distinguish between an algorithm for *evaluating* $M(n)$ at one given integer n, and an algorithm for *tabulating* $M(k)$ for all integers k in the interval $[1, n]$, which we may refer to simply as "tabulating $M(n)$". Several of the exact evaluation algorithms can be modified in an obvious way to give a tabulation algorithm with essentially the same time and space complexity.[3] This is not true of the Monte Carlo algorithms, which output an estimate of a single value $M(n)$ and give no information on $M(k)$ for $k \neq n$.

Regarding exact algorithms, our contributions are an extension of the previous numerical work on evaluation/tabulation of $M(n)$, and the development of an asymptotically faster (subquadratic) tabulation algorithm. Specifically, we can evaluate $M(k)$ for all $k \le n$ in time $O(n^2/L^{1/\sqrt{2}+o(1)})$, where

$$L = L(n) := \exp\left(\sqrt{\log n \log\log n}\right),$$

using $O(n)$ space. We present this algorithm (Algorithm 5) via a series of steps. First, we explain a straightforward algorithm (Algorithm 1) to evaluate $M(n)$ in time and space $O(n^2)$. This algorithm[4] was used by Brent and Kung [8]. Second, we show (Algorithm 2) that we can evaluate $M(n)$ given $M(n-1)$, using only $O(n)$ space and $O(n \log n)$ time.[5] This incremental approach naturally leads to a tabulation algorithm (Algorithm 3) that uses $O(n)$ space and $O(n^2 \log n)$ time. Finally, we refine the incremental approach to obtain a subquadratic tabulation algorithm (Algorithm 5): see Theorem 2.9 for the time bound.

For arguments $n \le 2^{30}$, our implementation of Algorithm 5 is slower than that of the best quadratic algorithm. This is a familiar phenomenon: for many other problems (e.g. multiplication of n-bit integers, or of $n \times n$ matrices) the asymptotically fastest known algorithm is not necessarily the fastest in practice.

We have tabulated $M(n)$ for $n \le 2^{30}$, using a variant of Algorithm 3. For confirmation of the numerical results we used Algorithm 1 (with segmentation and parallelisation) for various values of n, including $n = 2^k - 1$ for $k = 1, 2, \ldots, 30$.

The known exact (quadratic or subquadratic) evaluation/tabulation algorithms are too slow to go much past $n = 2^{30}$, so for larger n it is necessary to resort to approximate (Monte Carlo) methods. We give two Monte Carlo algorithms, which we call *Bernoulli* and *product*, for reasons that will be evident later (see §3). In each case, we avoid the problem of factoring large integers by using Bach's algorithm [2, 20] for generating random integers in factored form. As far as we are aware, this project represents the first time that the Bach algorithm for producing random factored numbers has been usefully implemented. The speed of the Monte Carlo algorithms depends mainly on the time required for testing the primality[6] of

[3] Space is measured in bits, and does not include any space required to store the output.

[4] The space requirement can be reduced by segmentation, see §2.1.

[5] The time bound can be improved, see Theorems 2.5–2.6 in §2.3.

[6] Or "probable" primality, since a small probability of error is acceptable in the context of a Monte Carlo computation, see §3.3.

large integers, which can be done much faster than factoring integers of comparable size [1, 28].

The paper is organized as follows. Section 2 is concerned with exact algorithms for evaluating/tabulating $M(n)$. After a brief overview of the sieve of Eratosthenes as a precursor to various ways of evaluating $M(n)$, we start with the method used by Brent and Kung [8]. We then show how to tabulate $M(n)$ in time $O(n^2 \log n)$ using an incremental algorithm. In fact, the time bound can be reduced slightly by using a result of Ford [15], as we show in Theorems 2.5–2.6 and Remark 2.7. In §2.4 we consider generating products in a multiplication in specific residue classes. We then show (in Theorem 2.9) that the incremental algorithm can be modified to tabulate $M(n)$ in time $O(n^2/L^{1/\sqrt{2}+o(1)})$. We remark that $\log n = L^{o(1)}$, so $\log n$ factors can be subsumed into the $o(1)$ term in the exponent of L.

Section 3 describes and compares our two Monte Carlo algorithms for estimating $M(n)$. We consider the variance in their estimates for a given number of independent random trials. Lemma 3.2 shows that, considering only the variance, the product algorithm is more accurate than the Bernoulli algorithm. This does not necessarily mean that it is preferable in practice, as factors such as the time per trial and space requirements need to be considered.

Finally, Section 4 contains numerical results (exact up to $n = 2^{30}-1$, and approximate up to $n = 2^{100,000,000}$), comments on implementations of the algorithms, and some conclusions regarding the asymptotic behaviour of $M(n)$.

Remark 1.1. In (3), the factor $(\log n)^c$ is asymptotically larger than the factor $(\log\log n)^{3/2}$. However, for small n, the second factor varies more rapidly than the first. Write $x := \log n$, $A = A(x) := x^c$, $B = B(x) := (\log x)^{3/2}$. Thus $\Phi(n) = AB$ and, taking logarithmic derivatives, we have $\Phi'/\Phi = A'/A + B'/B$. Now $|A'/A| < |B'/B|$ if $c/x < 3/(2x \log x)$, i.e. if $x < \exp(3/2c) \approx 3.7 \times 10^7$, or

$$n < \exp(\exp(3/2c)) \approx 2^{53,431,908}.$$

Consequently, our numerical results up to $n = 2^{100,000,000}$ barely extend to the region where the true asymptotic behaviour of $M(n)$ becomes evident.

2. Exact Evaluation of $M(n)$

Our model of computation is a standard random access machine with infinite, direct-access memory. Memory may be addressed at the bit-level or at the word level, and the word size is $\Theta(\log n)$ bits if n is the input size. We assume that arithmetic operations, memory access, and other basic operations take unit time. We count space in bits and do not include any space used to store the output.

2.1. **Sieve of Eratosthenes.** The algorithms for evaluating $M(n)$ resemble the sieve of Eratosthenes, the simplest implementation of which involves, for each $1 < k \le n^{1/2}$, removing the multiples of k from $(k, n]$. This naive implementation uses $O(n \log n)$ time and $O(n)$ space and finds all primes up to n. There is a large body of literature, both practical and theoretical, dealing with improvements and variations to this sieve. We refer to Helfgott [18] for a summary of the literature. Here, we highlight the aspects that are relevant for computing $M(n)$. In practice, we may be limited by a space constraint; lowering the space used by an algorithm may turn otherwise infeasible computations into feasible ones. For example, before marking off multiples of k in $(k, n]$, we may segment this interval into subintervals.

The asymptotic run-time remains unchanged so long as the "marking off" process is not doing "empty work", i.e. so long as $(n-k)/k$ is not small. Using this idea, the space bound may be reduced to $O(n^{1/2})$ with straightforward segmentation of the interval $[1, n]$. Helfgott [18] reduces the space requirement further by using Diophantine approximation to predict which integers less than $n^{1/2}$ have multiples in a given subinterval. This prediction process allows sieving on intervals of size $O(n^{1/3}(\log n)^{5/3})$ at no asymptotic cost in time [18, Main Theorem].

2.2. **Computing $M(n)$ Directly.** We can explicitly construct each product in a multiplication table and count the number of distinct products using Algorithm 1. The algorithm exploits the symmetry of the multiplication table.

Algorithm 1: Computing $M(n)$ directly

```
Input  : An integer $n$
Output: $M(n)$
Initialize a bit vector $A$ of length $n^2$ to 0.
for $1 \le i \le n$ do
    for $i \le j \le n$ do
        Set $A[ij] = 1$
return Hamming weight of $A$
```

The following lemma is obvious from counting the number of times the body of the inner loop is executed . We note that the area associated with the table is n^2.

Lemma 2.1. *Algorithm 1 computes $M(n)$ in time $O(n^2)$ and space $O(n^2)$.*

Algorithm 1 looks similar to the sieve of Eratosthenes (for finding the primes smaller than n^2), and many of the tricks that are known to speed up the latter may also be used to speed up Algorithm 1. The key difference is the stopping point for marking off multiples of i; Algorithm 1 only marks off through the nth multiple of i. Because of this early stopping point, Algorithm 1 has time bound $O(n^2)$, whereas the corresponding bound for a naive version of the sieve of Eratosthenes is $O(n^2 \log n)$.

The space bound of Lemma 2.1 can be reduced by modifying the algorithm. As discussed above, standard segmenting allows subintervals of size $O((n^2)^{1/2}) = O(n)$. By using Diophantine approximation the space bound could even be reduced to $O(n^{2/3}(\log n)^{5/3})$.

Suppose that it is possible to store all n^2 bits of the vector A. If the bit vector A associated with the computation of $M(n-1)$ is saved, then $M(n)$ may be computed in $O(n)$ additional time. We simply count how many bits are *not* set in $S = [A[n], A[2n], \ldots, A[n^2]]$, and increment $M(n-1)$ by that amount. Let the number of set bits in S be denoted by $\delta(n)$, so the number of unset bits is $n - \delta(n)$. We can compute $\delta(n)$ in $O(n)$ time, and we can compute $M(n)$ using

$$M(n) = M(n-1) + (n - \delta(n)) = \sum_{k=1}^{n} \big(k - \delta(k)\big) = \frac{n^2+n}{2} - \sum_{k=1}^{n} \delta(k). \tag{4}$$

§2.3 shows how to compute $\delta(n)$ almost as quickly, using only $O(n)$ space.

2.3. **Computing $M(n)$ Incrementally.** We compute $M(n)$ incrementally using (4). More precisely, we compute $\delta(k)$ for all $k \leq n$ where $\delta(n)$ counts the elements in $\{n, 2n, 3n, \ldots, n^2\} = \{mn : 1 \leq m \leq n\}$ that appear in the $(n-1) \times (n-1)$ multiplication table. If mn appears in the smaller multiplication table then it may be factored so that each factor is strictly less than n. If $m = ij$ and $n = gh$, then $mn = (ij)(gh) = (ih)(jg)$. If $ih < n$ and $jg < n$ then the product mn has already appeared in the table. Observe that $ih < n$ iff $i < g$ and $jg < n$ iff $j < h$. Thus, to compute $\delta(n)$, we need to count distinct products ij with $0 < i < g$ and $0 < j < n/g$ for each divisor g of n with $g \leq \sqrt{n}$. By counting the distinct products in the shape formed by the union of rectangles whose boundaries are determined by the divisors of n, we may compute $\delta(n)$.

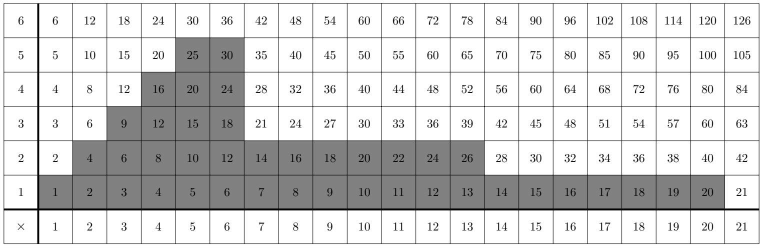
Algorithm 2: Computing $\delta(n)$

```
Input  : $D = [[d_0 = 1, n], \ldots, [d_{\ell-1}, n/d_{\ell-1}]]$, containing the ordered pairs of
         divisors of $n$, where $d_{\ell-1}$ is the largest divisor in $[1, \sqrt{n}]$.
Output: $\delta(n)$
Initialize counters $i = 1$ and $k = 0$
Initialize a bit vector $A$ of length $n$ to 0.
for $i < D[\ell-1][0]$ do
    if $i == D[k][0]$ then
        Increment $k$
    for $i \leq j < D[k][1]$ do
        Set $A[ij] = 1$
return Hamming weight of $A$
```

Remark 2.2. If the input to Algorithm 2 is missing one or more divisor pairs, then the output (Hamming weight of A) is a lower bound on $\delta(n)$.

Example 2.3. In Figure 1, the gray area corresponds to the products that Algorithm 2 constructs given the divisor pairs $2 \cdot 21$, $3 \cdot 14$, and $6 \cdot 7$ of 42.

Figure 1. The shape for computing $\delta(42)$



6	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114	120	126
5	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100	105
4	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84
3	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	57	60	63
2	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
×	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Algorithm 2 runs in time proportional to the area of the shaded region (which we call *the shape*). In general, an upper bound is $O(n \log n)$, obtained by noting that no product in a rectangle is larger than n, so the total area is bounded by the area under a hyperbola. A different upper bound is $O(n\tau(n))$, where $\tau(n)$

counts the divisors of n. This bound comes from the fact that, for each divisor of n, we construct a rectangle of area less than n. Both of these upper bounds can over-estimate. The $O(n \log n)$ bound over-estimates when n is not smooth, and the $O(n\tau(n))$ bound over-estimates when n is smooth [11]. The first bound may be used to show that $M(n)$ can be tabulated in time $O(n^2 \log n)$; this bound can be reduced to $o(n^2 \log n)$ by using deeper results on divisors, as in Theorem 2.6.

Following [14, 15], we let $\tau(n; y, z)$ be the number of divisors d of n which satisfy $y < d \le z$, and $\tau^+(n) = |\{k \in \mathbb{Z} : \tau(n, 2^k, 2^{k+1}) \ge 1\}|$. Lemma 2.4 (due to Ford) bounds the mean value of $\tau^+(n)$.

Lemma 2.4 (Ford [15, Corollary 5])**.** *If $c = 0.086\ldots$ is as above, then*

$$\frac{1}{n}\sum_{k\le n}\tau^+(k) = \Theta\Big(\frac{\log n}{\Phi(n)}\Big).$$

Theorem 2.5. *Algorithm 2 computes $\delta(n)$ in space $O(n)$ and in time $O(n\tau^+(n))$.*

Proof. By the above discussion concerning $\delta(n)$, the algorithm is correct. As i increases, the counter k keeps track of which rectangle boundary to use. The counter j is then bounded above by the appropriate divisor of n.

The space bound is obvious, since the vector A uses n bits.

For the time bound, recall that the run-time is proportional to an area, say $\mathcal{A}$. For each k, consider all the divisors of n in the interval $(2^k, 2^{k+1}]$. They all have the same bottom left corner, namely, the origin, and shapes range from $2^k \times n/2^k$ to $2^{k+1} \times n/2^{k+1}$. Hence they are all enclosed by a rectangle of shape $2^{k+1} \times n/2^k$ which has area $2n$. Thus we get an upper bound $\mathcal{A} \le 2n\tau^+(n)$. □

Clearly Algorithm 2 can be invoked repeatedly to tabulate $M(n)$. For reference we call this (tabulation algorithm) *Algorithm* 3.

Theorem 2.6. *Algorithm 3 tabulates $M(n)$ in space $O(n)$ and time*

$$O\left(\frac{n^2 \log n}{\Phi(n)}\right).$$

Proof. We compute $M(n)$ by evaluating $\delta(k)$ for $1 \le k \le n$. Using Theorem 2.5, the total run-time is

$$O\left(\sum_{k\le n} k\tau^+(k)\right) = O\left(n\sum_{k\le n}\tau^+(k)\right),$$

so the result now follows from Lemma 2.4. □

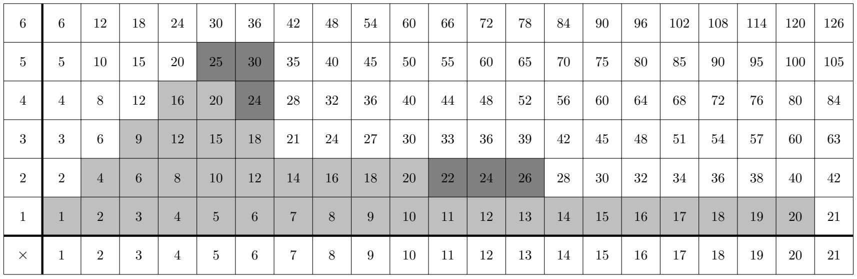
Remark 2.7. In view of Ford's result (2), the time bound given in Theorem 2.6 can be written as $O(M(n)\log n)$. We do not know how to give a direct proof of this without using Ford's results.

The space bound in Theorem 2.5 is for a naive implementation. It is not difficult to see that it can be reduced to $O(n^{1/2})$ with straightforward segmentation, or even to $O(n^{1/3}(\log n)^{5/3})$ via Diophantine approximation, as in [18]. Algorithm 3 represents an improvement by a factor of n in the naive storage cost and a significant improvement in run-time for the tabulation problem. If only a single evaluation is required, then Algorithm 1 may be faster. In practice, we observed that Algorithm 3 is competitive with Algorithm 1. This may be due to different implied constants in the big-O bounds, and because Algorithm 1 has a larger memory requirement, which can cause a deviation from the expected quadratic run time due to cache effects [19, Chapter 2]. In the next subsection we explain how generating products in specific residue classes can be used to speed up Algorithm 2.

2.4. **Working "modulo w".** We may generate products in a multiplication table in specific residue classes; this is akin to sieving with a wheel, and has two advantages. First, if w is the modulus, then the vector used in Algorithm 2 may be declared to be of size $\lfloor n/w \rfloor$ and unique products may be counted by residue class. Second, by not explicitly constructing small consecutive products, but simply counting them, we get a faster algorithm. In the following we illustrate these ideas with the examples $w = 1, 2$, and 6.

2.4.1. *Working "modulo* 1*".* If n is not prime, then the first row of the table contains the consecutive integers less than the largest nontrivial divisor of n. Store the number of consecutive integers and initialize the bit vector A so that the zero index is associated with the largest divisor of n. Iterate through each row of the multiplication table starting at the first entry greater than or equal to the largest divisor. Figure 2 shows the area that is considered in computing $\delta(42)$. The light gray products are all accounted for because the first row has 20 distinct products. We only construct the products greater than 20, which are shown in dark gray. This improvement reduces both the time and space requirements by a factor of $(1 - 1/p_1)$, where p_1 is the smallest prime factor of n. As a consequence, it is easy to see that $\delta(2p) = p - 1$ if (as usual) p is a prime.

Figure 2. The shape for computing $\delta(42)$ working modulo 1.

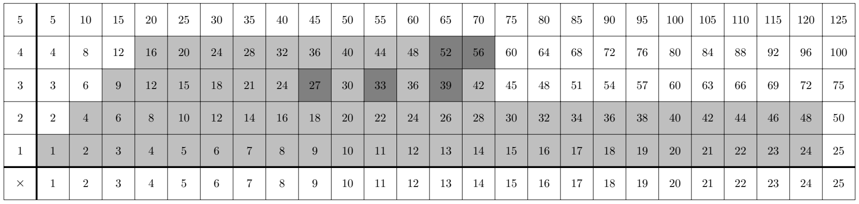


6	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114	120	126
5	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100	105
4	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84
3	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	57	60	63
2	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
×	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

2.4.2. *Working "modulo* 2*".* If n is composite and not of the form $2p$ for p a prime, then its shape has nontrivial entries in the first two rows. Create a bit vector associated with odd numbers. The first row is indexed by consecutive odd numbers up to some bound. Either the first row or the second row will contain the bound for the consecutive even numbers that are stored. For rows associated with an odd multiplier, start with the lower bound associated with the odd vector and iterate through the table creating only the odd entries. For the even vector, consider even rows and the even numbers in the odd rows. This reduces the time by reducing area although the overhead in setting up the loops to iterate through the table in the specified manner is higher. More importantly, it reduces the memory requirement. By splitting the products into residue classes modulo 2, we require half the storage. The above discussion also makes it easy to see that $\delta(3p) = p - 1 + \lfloor (p - 1)/2 \rfloor$.

Figure 3 shows the area that is considered in computing $\delta(75)$ modulo 2. The products in light gray are accounted for by a counting argument and the products in dark gray are constructed. That is, the bit vector storing even numbers starts at 50, and then the even products 52 and 56 in dark gray are constructed. Thus, there are $24+2 = 26$ unique even products. The bit vector storing odd numbers starts at 25, and constructs the products $27, 33, 39$. Therefore, there are $12 + 3 = 15$ unique odd products, and $\delta(75) = 26 + 15 = 41$.

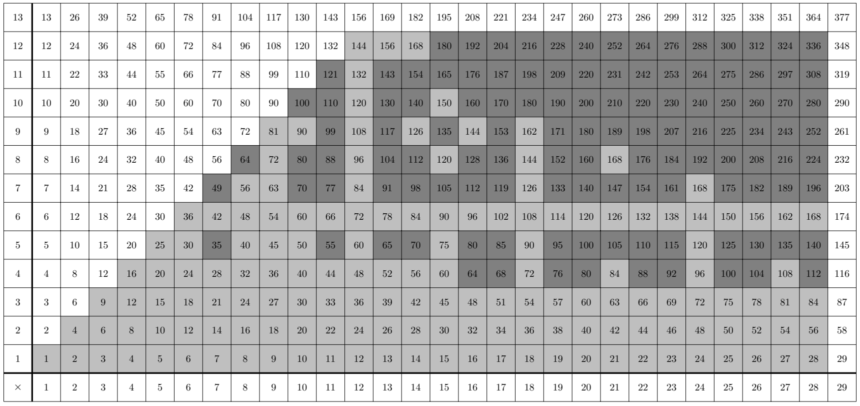
Figure 3. The shape for computing $\delta(75)$ working modulo 2.



5	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100	105	110	115	120	125
4	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	96	100
3	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	57	60	63	66	69	72	75
2	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
×	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

2.4.3. *Working "modulo 6"*. A naive invocation of Algorithm 2 to compute $\delta(377)$ requires the construction of 270 products. By constructing products in residue classes modulo 6 only 119 products need to be constructed. In Figure 4, we see that the sixth row tells us there are 28 consecutive multiples of 6. Therefore, we only need to construct products $0 \pmod 6$ that are greater than 168. Similarly, the third row tells us that there are 14 consecutive numbers $3 \pmod 6$. Therefore, we only construct products $3 \pmod 6$ that are greater than 84. The second row tells us that we only need to construct products greater than 56 when we deal with $2, 4 \pmod 6$ cases. Finally, the first row tells us that we need to construct products greater than 28 for the $1, 5 \pmod 6$ cases.

Figure 4. The shape for computing $\delta(377)$ working modulo 6.



13	13	26	39	52	65	78	91	104	117	130	143	156	169	182	195	208	221	234	247	260	273	286	299	312	325	338	351	364	377
12	12	24	36	48	60	72	84	96	108	120	132	144	156	168	180	192	204	216	228	240	252	264	276	288	300	312	324	336	348
11	11	22	33	44	55	66	77	88	99	110	121	132	143	154	165	176	187	198	209	220	231	242	253	264	275	286	297	308	319
10	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200	210	220	230	240	250	260	270	280	290
9	9	18	27	36	45	54	63	72	81	90	99	108	117	126	135	144	153	162	171	180	189	198	207	216	225	234	243	252	261
8	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128	136	144	152	160	168	176	184	192	200	208	216	224	232
7	7	14	21	28	35	42	49	56	63	70	77	84	91	98	105	112	119	126	133	140	147	154	161	168	175	182	189	196	203
6	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114	120	126	132	138	144	150	156	162	168	174
5	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100	105	110	115	120	125	130	135	140	145
4	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	96	100	104	108	112	116
3	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	57	60	63	66	69	72	75	78	81	84	87
2	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52	54	56	58
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
×	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

It is possible to create rules for the evaluation of $\delta(mp)$ via counting arguments. We count consecutive products in residue classes modulo $w = \text{lcm}(1, 2, 3, \ldots, m-1)$. The third author created a website [26] that may be used to count the products constructed and display the shape associated with a $\delta(n)$ computation when using Algorithm 2 naively, or with a modulus of $w = 1, 2, 6, 12, 60$, or 120.

2.5. **Subquadratic Tabulation.** Recall that if all n^2 bits of A can be held at once in Algorithm 1, then tabulation and evaluation are essentially the same problem. We apply this idea to computing $\delta(n)$. Consider the use of Algorithm 2 in computing $\delta(6 \cdot 7)$, $\delta(6 \cdot 9)$, $\delta(6 \cdot 11)$, and $\delta(6 \cdot 13)$. The divisor list for each of these is of the form $[1, 6 \cdot k], [2, 3 \cdot k], [3, 2 \cdot k]$, and $[6, k]$ for $k = 7, 9, 11, 13$. One shape is always a subset of the next shape and so the set of distinct products in each shape is a subset of the next such set. Rather than think of four independent computations, we consider the one computation of $\delta(6 \cdot 13)$. Unlike in Algorithm 2 where the bit vector storing distinct products is populated by a row of the multiplication table, we will populate the bit vector by incrementally shifting the end-points of the rectangles. While computing $\delta(6 \cdot 13)$ we can "learn" $\delta(6 \cdot 7)$, $\delta(6 \cdot 9)$, and $\delta(6 \cdot 11)$. Instead of computing $\delta(6 \cdot 9)$ from the beginning, we use the computation of $\delta(6 \cdot 7)$ and only account for the new products that may arise.

In general, this requires that we tabulate $\delta(n)$, for those n that have similar shapes. For a fixed m and primes $p \approx q$, the divisor lists of mp and mq are very similar. In particular, if both primes are larger than m, the first entries in the divisor lists correspond only to the divisors of m. If $p < q$, we may re-use the bit vector from computing $\delta(mp)$ to compute $\delta(mq)$. All we need to account for are the new products that appear as the corresponding rectangles are shifted.

Algorithm 4: Computing $\delta(mq)$ given $\delta(mp)$ for primes p, q ($m < p < q$).

```
Input  : A bit vector $A$ of length $mq$ with weight $w$ containing the products
         from computing $\delta(mp)$. The divisor lists for $mp$ and $mq$:
         $D_p = [[d_0 = 1, mp], [d_1, mp/d_1], \ldots]$ and
         $D_q = [[d_0 = 1, mq], [d_1, mq/d_1], \ldots]$ both of length $\ell$.
Output: $\delta(mq)$
Initialize counters $i = 1$ and $k = 0$
for $i < D_p[\ell - 1][0]$ do
    if $i == D_p[k][0]$ then
        Increment $k$
    for $D_p[k][1] \leq j < D_q[k][1]$ do
        if $A[ij] == 0$ then
            Set $A[ij] = 1$
            Increment $w$
return $w$
```

Lemma 2.8. *If $mq \leq n$, then Algorithm 4 computes $\delta(mq)$ in time $O(md(q) \log n)$, where $d(q) := q - p$.*

Proof. There are $O(m \log n)$ individual products to check per unit shift. □

The benefit of Algorithm 4 over Algorithm 2 is that, while computing $\delta(mq)$, we learn $\delta(mp)$ for all $p < q$. In computing $M(n)$, we may compute $\delta(mq)$ at a cost of $O(n \log n)$, but in the process we learn $\delta(mp)$ for all prime p, $m < p < q$, for no additional cost.

Theorem 2.9. *Algorithms 2 and 4 may be combined to tabulate $M(n)$ in time $O(n^2/L^{1/\sqrt{2}+o(1)})$, where $L = L(n) := \exp\left(\sqrt{\log n \log \log n}\right)$.*

Proof. Let γ be a real parameter with $0 < \gamma < 1$, to be chosen later. We split the integers $k \le n$ into two classes. The first consists of k that are L^γ-smooth, that is, all prime factors of k are at most L^γ. There are $n/L^{1/(2\gamma)+o(1)}$ such numbers k, as $n \to \infty$, see [10]. For these values of k we compute $\delta(k)$ via Algorithm 2, accounting for a run-time of $O(n^2/L^{1/(2\gamma)+o(1)})$. The second class consists of those k that are not L^γ-smooth; write such k as mq, where q is the largest prime factor of k, so that $q > L^\gamma$. Since $k \le n$, the pairs (m, q) that can arise here have $m < n/L^\gamma$. For each such pair (m, q) take the largest prime Q with $mQ \le n$ and compute $\delta(mQ)$ using Algorithm 4, so learning $\delta(mq)$ for all primes $q \le Q$, and in particular, for all primes q with $L^\gamma < q \le Q$. For each m the run-time is $O(n \log n)$ by Lemma 2.8, so the total run-time for all such values of m is $O(n^2/L^{\gamma+o(1)})$. These two computations are balanced when $\gamma = 1/\sqrt{2}$, proving the theorem. □

For reference we let *Algorithm* 5 be the algorithm defined by the above proof.

3. Monte Carlo Estimations

If n is too large for the exact computation of $M(n)$ to be feasible, we can resort to Monte Carlo estimation of $M(n)$. In the following we describe two different Monte Carlo algorithms, which we call the *Bernoulli* and *product* algorithms. In the descriptions of these two algorithms, we assume that n is fixed, and p denotes a probability (not a prime number).

3.1. **The Bernoulli Algorithm.**

We perform a sequence of $T \ge 2$ trials, where each trial involves choosing a random integer $z \in [1, n^2]$. The integers z are assumed to be independent and uniformly distributed. For each z, we count a *success* if z appears in the $n \times n$ multiplication table, i.e. if z can be written as $z = xy$, where $1 \le x \le y \le n$. Let S be the number of successes after T trials. Since we are performing a sequence of T Bernoulli trials with probability of success $p = M(n)/n^2$, the expected number of successes is $\mathbb{E}(S) = pT$, and the variance is $\mathbb{V}(S) = \mathbb{E}((S - pT)^2) = p(1-p)T$. Thus, an unbiased estimate of $M(n)/n^2$ is given by $\widehat{p} = S/T$, and the variance of this estimate is $p(1-p)/T$. For large T the error $M(n)/n^2 - S/T$ is asymptotically normally distributed. By the "law of the iterated logarithm" [21], this error is almost surely $O((T^{-1} \log\log T)^{1/2})$ as $T \to \infty$.

Remark 3.1. In a practical computation, p is unknown, but an unbiased estimate of the variance of the error is $\widehat{p}(1-\widehat{p})/(T-1)$, where the denominator $T-1$ takes into account the loss of one degree of freedom in using the sample mean $\widehat{p}$ instead of the population mean p. This is known as *Bessel's correction*, and was used by Gauss [17] as early as 1823.

3.2. **The Product Algorithm.**

In this algorithm, each trial takes $z = xy$, where x and y are independently and uniformly distributed integers in $[1, n]$. Thus, z is guaranteed to appear in the $n \times n$ multiplication table. Let $\nu = \nu(z) \ge 1$ denote the number of times that z appears in the table. The probability that a trial samples z is $\nu(z)/n^2$. Thus, $\mathbb{E}(1/\nu) = M(n)/n^2 = p$ (where p is as in the Bernoulli algorithm). Consider a sequence of T independent trials, giving values $\nu = \nu_1, \ldots, \nu_T$. An unbiased estimate of $M(n)/n^2$ is given by $E := T^{-1} \sum_{1 \le j \le T} 1/\nu_j$, and the variance of this estimate is $V := T^{-1}\mathbb{E}((\nu^{-1} - p)^2)$. Lemma 3.2 shows that, for the same values of T and n, the variance in the estimate of $M(n)/n^2$ given by the product algorithm is no larger than that given by the Bernoulli algorithm.

Lemma 3.2. *If V is the variance of the estimate E after T trials of the product algorithm, then $V \le p(1-p)/T$.*

Proof. Using $p = \mathbb{E}(\nu^{-1})$, we have

$$V = T^{-1}\mathbb{E}((\nu^{-1} - p)^2) = T^{-1}(\mathbb{E}(\nu^{-2}) - p^2).$$

Since ν is a positive integer, $\nu^{-2} \le \nu^{-1}$, and $\mathbb{E}(\nu^{-2}) \le \mathbb{E}(\nu^{-1}) = p$. It follows that $V \le T^{-1}(p - p^2)$, as desired. □

Remark 3.3. It is easy to see that equality holds in Lemma 3.2 only in the trivial case $n = 1$. From Ford's result (2), we have $TV = O(1/\Phi(n))$ as $n \to \infty$.

An unbiased estimate of the variance of the error for the product algorithm in terms of computed quantities is $\sum_{1\le j\le T}(\nu_j^{-1} - E)^2/(T(T-1))$, see Remark 3.1.

3.3. **Avoiding Factorization via Bach/Kalai.** For the Bernoulli algorithm, we have to determine if an integer $z \in [1, n^2]$ occurs in the $n \times n$ multiplication table. Equivalently, we have to check if z has a divisor d satisfying $z/n \le d \le n$. A straightforward algorithm for this would first find the prime power factorization of z, then attempt to construct a divisor d in the interval $[z/n, n]$, using products of the prime factors of z.

Similarly, for the product algorithm, we have to count the number of divisors d of xy in the interval $[xy/n, n]$. A straightforward algorithm for this would first find the prime power factorizations of x and y.

To avoid having to factor the random integers z (or x and y) occurring in the Bernoulli (or product) algorithms, we can generate random integers *along with their prime power factorizations*, using the algorithms of Bach [2] or Kalai [20]. This is much more efficient, on average, than generating random integers and then attempting to factor them, since the integer factorization problem is not known to be solvable in polynomial time and is time consuming in practice for many inputs.

The algorithm described by Bach, specifically his "Process R", returns an integer x uniformly distributed in the interval $(N/2, N]$, together with the prime power factorization of x. Using Bach's algorithm, which we call "procedure R", it is easy to give a recursive procedure B which returns x uniformly distributed in the interval $[1, N]$, together with the prime power factorization of x. For details see Algorithm 6. The following comments on the complexity of Bach's algorithm also apply to procedure B.

The expected running time of Bach's algorithm is dominated by the time for primality tests.[7] Bach's algorithm requires, on average, $O(\log N)$ primality tests. The AKS deterministic primality test [1, 22] requires $(\log N)^{O(1)}$ bit-operations, so overall Bach's algorithm has average-time complexity $(\log N)^{O(1)}$. In our implementation, we replaced the AKS primality test by the Miller–Rabin Monte Carlo test [9, 23, 24, 28], which is much faster, at the cost of a small probability of error.[8] A small probability of an error (falsely claiming that a composite integer is prime) is

[7] More precisely, Bach's algorithm requires prime power tests, but it is relatively easy to check if an integer is a perfect power (see Bernstein [4]), so primality tests and prime power tests have (on average) almost the same complexity. Also, it is possible to modify Bach's algorithm so that only primality (not prime power) tests are required. Thus, we ignore the distinction between primality tests and prime power tests.

[8] The probability of error can be reduced to $\le 4^{-k}$ by repeating the test k times with independent random inputs, see [28].

Algorithm 6: Modification of Bach's algorithm

```
procedure $R(N)$
Input  : A positive integer $N$
Output: A random integer $x \in (N/2, N]$ and its prime power factorization
Details omitted: see Bach [2, "Process R", pg. 184]
end procedure $R$
procedure $B(N)$
Input  : A positive integer $N$
Output: A random integer $x \in [1, N]$ and its prime power factorization
if $N == 1$ then
    return 1
generate random real $u$ uniformly distributed in $[0, 1)$
if $u < \lfloor N/2 \rfloor / N$ then
    return $B(\lfloor N/2 \rfloor)$
else
    return $R(N)$
end procedure $B$
```

acceptable when the overall computation is a Monte Carlo estimation. Such errors will have a negligible effect on the final result, assuming that the number of trials is large.

Kalai [20] gave an algorithm with the same inputs and outputs as our modification (procedure B) of Bach's algorithm, but much simpler and easier to implement. The disadvantage of Kalai's algorithm is that it is asymptotically slower than Bach's, by a factor of order $\log N$. More precisely, Kalai's algorithm requires, on average, of order $(\log N)^2$ primality tests, whereas procedure B requires of order $\log N$ prime power tests. We implemented both algorithms using Magma [5], and found that, as expected, Kalai's algorithm was slower than procedure B for N sufficiently large. With our implementations[9], the crossover point was $N \approx 2^{45}$. For $N = 2^{100}$, procedure B was faster by a factor of about 2.2.

4. Implementations and Results

We used several independent implementations of Algorithm 1 (with segmentation), and three independent implementations of Algorithm 3 in three different languages: C, C++, and Sage.[10] The published exact computations in [8] are of the form $M(2^n - 1)$ for $1 \le n \le 17$. In Table 1, we include $18 \le n \le 30$. The entries in Table 1 were computed independently using both Algorithm 1 and Algorithm 3. No discrepancies were found.[11] Timing comparisons are difficult as different (time-shared) computer systems were used, but we estimate that Algorithm 3 was about three times faster than Algorithm 1 for $n = 30$.

[9] Further details concerning our implementations, and approximations/optimizations valid for very large N, may be found in [6, 7].

[10] We thank Paul Zimmerman for verifying some of our results using Sage.

[11] The entries given in OEIS A027417 differ by one because they include the zero product.

k	$M(2^k - 1)$	k	$M(2^k - 1)$
18	14081089287	25	209962593513291
19	55439171530	26	830751566970326
20	218457593222	27	3288580294256952
21	861617935050	28	13023772682665848
22	3400917861267	29	51598848881797343
23	13433148229638	30	204505763483830092
24	53092686926154		

Table 1. Extension of the Brent-Kung computation

A table of $M(k \cdot 2^{10})$ for $1 \leq k \leq 2^{20}$ was computed by the third and fourth authors [27]. The computation used a wheel modulus approach as described §2.4 with $w = 60$. The computation took about 7 weeks on Butler University's BigDawg cluster which has 32 Intel Xeon E5-2630 processors (a total of 192 cores). Table 2 shows the time (in seconds) to compute $\delta(n)$ for all $n \in (10^8, 10^8 + 10^3]$ on an Intel i7-4700 with 16GB RAM, using various values of the modulus w. It can be seen that using larger moduli provides a significant speedup (at the cost of increased program complexity).

Algorithm	time (s)
Algorithm 2	909
(mod 1)	302
(mod 2)	184
(mod 6)	106
(mod 12)	85
(mod 60)	59

Table 2. Runtime comparison

Algorithm 1, implemented in C, ran on the ARCS computer system at the University of Newcastle, Australia. The computer nodes used were a mixture of 2.2 GHz Intel Xeon 3 and 2.6GHz Intel Xeon 4.

We now consider Monte Carlo algorithms for approximating $M(N)$, where $N := 2^n - 1$. First consider the case $n = 30$, $N = 2^{30} - 1$, for which we know the exact value $M(N) = 204505763483830092$ from our deterministic computations. Taking $T = 10^6$ trials of the "product" Monte Carlo algorithm, we estimate $M(N)/N^2 = 0.17750$, whereas the correct value to 5 decimals (5D) is $M(N)/N^2 = 0.17738$. The variance estimate here is $V = 2.873 \times 10^{-8}$, so $\sigma := V^{1/2} \approx 0.00017$. Thus, the Monte Carlo estimate is as accurate as predicted from the standard deviation σ. The same number of trials with the Bernoulli algorithm gives variance 1.459×10^{-7}, larger by a factor of about five. Thus, the product algorithm is more efficient (other things being equal), as predicted by Lemma 3.2. In practice the comparison is not so straightforward, because the product algorithm requires checking more divisors (on average) than the Bernoulli algorithm, and has a larger space requirement.

The results of some Monte Carlo computations are given to 4D in Table 3. For $n > 10^4$ we used an approximation described in [6] to avoid dealing with n-bit integers (essentially by using a logarithmic representation). We used the product algorithm (mainly for $n < 10^6$) and the Bernoulli algorithm (mainly for

$n \geq 10^6$), combined with Bach's algorithm (described in Section 3). The Bernoulli algorithm was preferred for $n \geq 10^6$ because of its smaller space requirements. Kalai's algorithm was used for confirmation (mainly for $n \leq 100$).

The second column of Table 3 gives an estimate of $M(N)/N^2$, and the last column gives the normalized value $(N^2/M(N))/\Phi(N)$. By Ford's result (2), this should be bounded away from 0 and ∞ as $n \to \infty$. The third column gives $10^4\sigma$, where σ^2 is an estimate of the variance of the corresponding entry in the second column. Because of the factor 10^4, this corresponds to units in the last place (ulps) for the second column. Since the entries in the third column are bounded by 0.12, the entries in the second column are unlikely to be in error by more than 0.7 ulp. Similarly, the entries in the last column of the table are unlikely to be in error by more than 1 ulp.[12] The entries for $n \leq 30$ may be verified (up to the predicted accuracy) using the exact results of Table 1.

n	$M(N)/N^2$	$10^4\sigma$	$\frac{\text{trials}}{10^8}$	$\frac{N^2/M(N)}{\Phi(N)}$
	$(N = 2^n - 1)$			
20	0.1987	0.12	2	0.9414
30	0.1774	0.02	100	0.8213
40	0.1644	0.02	100	0.7549
50	0.1552	0.02	100	0.7112
10^2	0.1311	0.02	100	0.6068
10^3	0.0798	0.02	100	0.4264
10^4	0.0517	0.01	100	0.3435
10^5	0.0348	0.06	2	0.2958
10^6	0.0240	0.05	10	0.2652
10^7	0.0170	0.05	6.7	0.2432
10^8	0.0121	0.10	1.32	0.227

Table 3. Monte Carlo computations

It has not been shown that the numbers in the last column of Table 3 should tend to a limit as $N \to \infty$. Ford's result (2) shows that the lim sup and lim inf are finite and positive, but not that a limit exists. A non-rigorous extrapolation of our experimental results, described in more detail in [6, 7], suggests that the limit (if it exists) is about 0.12. Clearly convergence is very slow. Perhaps this is to be expected, given that $\Phi(n)$ grows very slowly.

In some similar problems the corresponding limit does not exist. For example, let $S(x)$ be the number of $n \leq x$ such that the number of divisors of n is at least $\log x$. Norton [25] showed that there are positive constants c_1, c_2 with $c_1 < R(x) < c_2$ for x sufficiently large, where $R(x) = S(x)x^{-1}(\log x)^c(\log\log x)^{1/2}$. Later, Balazard, *et al.* [3] showed that $\lim_{x\to\infty} R(x)$ does not exist. Thus, it would not be too surprising if $\lim_{N\to\infty} N^2/(M(N)\Phi(N))$ failed to exist. However, we have not detected any numerical evidence for oscillations in the last column of Table 3, so we would expect the lim inf and lim sup to be close, even if unequal.

References

[1] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena, *PRIMES is in P*, Ann. of Math. (2) **160** (2004), 781–793.

[12] Table 3 is extended to $n = 5 \times 10^8$ (but with lower accuracy) in [6, 7].

[2] Eric Bach, *How to generate factored random numbers*, SIAM J. Comput. **17** (1988), 179–193.

[3] M. Balazard, J. L. Nicolas, C. Pomerance, and G. Tenenbaum, *Grandes déviations pour certaines fonctions arithmétiques*, J. Number Theory **40** (1992), 146–164 (in French).

[4] Daniel J. Bernstein, *Detecting perfect powers in essentially linear time*, Math. Comp. **67** (1998), 1253–1283.

[5] Wieb Bosma, John Cannon, and Catherine Playoust, *The Magma algebra system. I. The user language*, J. Symbolic Comput. **24** (1997), 235–265.

[6] Richard P. Brent, *The multiplication table, and random factored integers*, slides from a seminar given at Hong Kong Baptist University, Hong Kong, Feb. 6, 2015. `https://maths-people.anu.edu.au/~brent/pd/multiplication-HK.pdf`.

[7] Richard P. Brent, *Algorithms for the multiplication table*, slides from a seminar given at CARMA, Newcastle, NSW, May 29, 2018. `https://maths-people.anu.edu.au/~brent/pd/multiplication-CARMA.pdf`.

[8] Richard P. Brent and H. T. Kung, *The area-time complexity of binary multiplication*, J. Assoc. Comput. Mach. **28** (1981), 521–534. Corrigendum: *ibid* **29** (1982), 904.

[9] Ronald J. Burthe, Jr., *Further investigations with the strong probable prime test*, Math. Comp. **65** (1996), 373–381.

[10] E. R. Canfield, P. Erdős, and C. Pomerance, *On a problem of Oppenheim concerning 'Factorisatio Numerorum'*, J. Number Theory **17** (1983), 1–28.

[11] Richard Crandall and Carl Pomerance, *Prime Numbers: A Computational Perspective*, second edition, Springer, New York 2005.

[12] Paul Erdős, *Some remarks on number theory*, Riveon Lematematika **9** (1955), 45–48 (in Hebrew).

[13] Paul Erdős, *An asymptotic inequality in the theory of numbers*, Vestnik Leningrad. Univ. **15** (1960), no. 13, 41–49 (in Russian).

[14] Paul Erdős and G. Tenenbaum, *Sur la structure de la suite des diviseurs d'un entier*, Ann. Inst. Fourier (Grenoble) **31** (1981, no. 1, ix, 17–37 (in French).

[15] Kevin Ford, *The distribution of integers with a divisor in a given interval*, Ann. of Math. (2) **168** (2008), no. 2, 367–433.

[16] Kevin Ford, *Integers with a divisor in* $(y, 2y]$, Anatomy of integers, CRM Proc. Lecture Notes, vol. 46, Amer. Math. Soc., Providence, RI, 2008, pp. 65–80.

[17] C. F. Gauss, Theoria combinationis observationum erroribus minimis obnoxiae, *Carl Friedrich Gauss Werke*, Bd. 4, Göttingen, 1873, 1–26 (in Latin).

[18] Harald A. Helfgott, *An improved sieve of Eratosthenes*, Math. Comp. **89** (2020), 333-350.

[19] John L. Hennessy and David A. Patterson, *Computer Architecture: a Quantative Approach*, fifth edition, Elsevier, 2012.

[20] Adam Kalai, *Generating random factored numbers, easily*, J. Cryptology **16** (2003), 287–289.

[21] Aleksandr Y. Khinchin, *Über einen Satz der Wahrscheinlichkeitsrechnung*, Fundamenta Mathematicae **6** (1924), 9–20 (in German).

[22] H. W. Lenstra jr. and Carl Pomerance, *Primality testing with Gaussian periods*, J. European Math. Soc. **21** (2019), 1229–1269.

[23] Jared D. Lichtman and Carl Pomerance, *Improved error bounds for the Fermat primality test on random inputs*, Math. Comp. **87** (2018), 2871–2890.

[24] Gary L. Miller, *Riemann's hypothesis and tests for primality*, J. Comp. System Sci. **13** (1976), 300–317.

[25] Karl K. Norton *On the number of restricted prime factors of an integer, I*, Illinois J. Math. **20** (1976), 681–705.

[26] David Purdum, *Multiplication table*, `https://rutrum.github.io/multiplication-table/`.

[27] David Purdum and Jonathan Webster, `http://blue.butler.edu/~jewebste/Mn2pow30.txt`.

[28] Michael O. Rabin, *Probabilistic algorithm for testing primality*, J. Number Theory **12** (1980), 128–138.