Journal of Integer Sequences, Vol. 28 (2025),
Article 25.8.5

On One of Erdős' Problems—An Efficient Search for Benelux Pairs

Christian Hercher
Institut für Mathematik
Europa-Universität Flensburg
Auf dem Campus 1c
24943 Flensburg
Germany
christian.hercher@uni-flensburg.de

Abstract

Erdős asked for positive integers $m < n$ such that m and n have the same set of prime factors, $m+1$ and $n+1$ have the same set of prime factors, and $m+2$ and $n+2$ have the same set of prime factors. No such integers are known. If one relaxes the problem and only considers the first two conditions, an infinite series of solutions is known: $m = 2^k - 2$, $n = (m+1)^2 - 1 = 2^k \cdot m$ for all integers $k \geq 2$. One additional solution is also known: $m = 75 = 3 \cdot 5^2$ and $n = 1215 = 3^5 \cdot 5$ with $m+1 = 76 = 2^2 \cdot 19$ and $n+1 = 1216 = 2^6 \cdot 19$. No other solutions with $n < 2^{32} \approx 4.3 \cdot 10^9$ were known.

In this paper, we discuss an efficient algorithm to search for such integers, also known as Benelux pairs, using sieving and hashing techniques. Using highly parallel algorithms on a modern consumer GPU, we confirmed the previously known results within a minute of computing time. Additionally, we expanded the search space by a factor of more than 2^{16} and found no further solutions different from the infinite series given above up to $1.4 \cdot 10^{12} > 2^{40}$.

For the analogous problem of integers $m < n$ with m and $n+1$ having the same set of prime factors and $m+1$ and n having the same set of prime factors, the situation is very similar: An infinite series and one exceptional solution with $n \leq 2^{22}+2^{12} \approx 4.2 \cdot 10^6$ were known. We prove that there are no other exceptional solutions with $n < 1.4 \cdot 10^{12}$.

1 Introduction

In 1996, Erdős [2] remarked that:

> Balsubramanian called to my attention another old problem of mine which I had forgotten. Can there ever be two distinct integers m and n for which for $0 \leq i \leq 2$ the integers $m+i$ and $n+i$ have the same prime factors? The answer is Yes for a more restricted problem. For m, n and $m+1, n+1$ we can take $m = 2(2^k - 1), n+1 = (m+1)^2$. Then m and n also have the same prime factors. Are there other integers m, n and $m+1, n+1$ with the same prime factors?

Note that we changed variable names in this quote for improved readability of this paper. Clearly, for all $k \geq 1$,

- the terms $m = 2 \cdot (2^k - 1)$ and $n = (m+1)^2 - 1$ are positive integers with $m < n$,
- m and $n = (m+1)^2 - 1 = m(m+2) = 2 \cdot (2^k - 1) \cdot 2^{k+1}$ have the same prime factors,
- and $m+1$ and $n+1 = (m+1)^2$ have the same prime factors, too.

Thus, all these pairs are solutions to the restricted problem. But, since $m + 2 = 2^{k+1}$ is a power of two and $n + 2 = (2^k - 1) \cdot 2^{k+2} + 2 \equiv 2 \pmod 4$ is not, none of these pairs is a solution to the original problem.

Guy [3, B19] lists the restricted problem, too. Here, we are told that Mąkowski found the exceptional solution $m = 3 \cdot 5^2$, $n = 3^5 \cdot 5$ with $m + 1 = 2^2 \cdot 19$ and $n + 1 = 2^6 \cdot 19$. But $n + 2 = 7 \cdot 11$ and $m + 2 = 1217$ is a prime. No other exceptional solution is known.

In 2011, the mathematical competition Benelux Olympiad [4] coined in their Problem 1 the term Benelux pair for positive integers $m < n$ when m and n, as well as $m+1$ and $n+1$, share the same prime factors.

The original problem from Erdős cited above is listed as Number #850 on Bloom's list [1] of Erdős' problems.

A variant of this problem is mentioned in Guy [3, B19]. This variant asks for two positive integers $m < n$ such that now m and $n+1$ should share the same prime factors, and $m+1$ and n do as well. The situation for this variant is nearly the same as with Benelux pairs. An infinite series of solutions is given by $m = 2^k + 1$, $n = m^2 - 1$. With this setting m and $n + 1 = m^2$ have the same prime factors, and $m + 1 = 2^k + 2 = 2 \cdot (2^{k-1} + 1)$ and $n = (m-1)(m+1) = 2^k \cdot 2 \cdot (2^{k-1} + 1)$ have the same prime factors. There is also one known exceptional solution. According to Guy [3], if $m = 5 \cdot 7$, $n + 1 = 5^4 \cdot 7$, then $n = 2 \cdot 3^7$, $m + 1 = 2^2 \cdot 3^2$. In an analogous way, this paper uses the term *Benelux pair of second kind* for positive integers $m < n$ with m and $n+1$ having the same prime factors, and $m+1$ and n having the same prime factors.

1.1 Computational verification and contributions from Schott, Ehrenstein, Nomoto, and Wasserman

According to sequence A343101 in the On-Line Encyclopedia of Integer Sequences, Schott computed all Benelux pairs (m, n) with n up to 2^{30}. Ehrenstein has expanded the list up to $n < 2^{32}$.

For the variant of the problem, Nomoto and Wasserman computed all Benelux pairs of second kind in the OEIS sequence A088966 up to $n \leq 2^{22} + 2^{12}$.

1.2 Contribution of this work

In this paper, we present an algorithm that computes all Benelux pairs and all Benelux pairs of second kind up to $m < n < S$ with a time complexity of $\mathcal{O}(S \log S)$ and a space complexity of $\mathcal{O}(S)$. Since the desktop PC used for the computation had a limited amount of RAM, this algorithm is impractical for larger values of S. Therefore, we provide a second algorithm that runs in $\mathcal{O}\left(\frac{1}{s} \cdot S^2\right)$ time and $\mathcal{O}(s)$ space, assuming that $s \in \Omega(\log S \log\log S)$. This second algorithm has the advantage that the parts that need the most computation can be highly parallelized. Thus, we used the GPU of the PC in hand to extend the search for such pairs by a large amount.

1.3 Outline

In Section 2, we construct and discuss in detail the two aforementioned algorithms before we conclude in Section 3 with a discussion of further improvements for high parallel computing, and provide the results of our computation. The source code of the CUDA/C++ program written for this purpose can be found at `https://github.com/ChristianHercher/Erdos850`.

2 Algorithms

2.1 Finding Benelux pairs quickly

Both problems—finding Benelux pairs of first and second kind—consider pairs of positive integers and their respective prime factors, while multiplicities are ignored. Thus, a good way to start is to calculate for every positive integer $n \leq S$ the set of primes that divide n. A first insight into these problems is that we do not have to know which primes exactly divide n, but rather only whether two such sets are identical or not. Hence, we do not need to compute the whole prime factorization of n if we only compute a number $r(n)$ with the property that $r(n) = r(m)$, if and only if n and m have the same prime factors. Such a function r is the *radical* of n: $\operatorname{rad}(n) := \prod_{p|n} p$. And this function has the great advantage that if we want to determine its value for all elements in a large interval, it can be computed efficiently via sieving.

This can be achieved using Algorithm 1, a variant of the well-known sieve of Eratosthenes: First, for every integer n in the half-open interval $[\text{start}, \text{start}+\text{length})$ we initialize the value $r[n]$ as $r[n] := n$. Then we subsequently divide this value by all primes p that have a multiplicity of > 1 in the prime factorization of n—and do this multiple times until the prime p remains with multiplicity 1 in the resulting quotient. To do this, we handle each prime p one at a time. The largest prime we have to consider is at maximum $\sqrt{\text{start} + \text{length} - 1}$ since for all larger primes p, the integer p^2 cannot be a divisor of any number in the considered interval. For all primes p up to this limit, we identify all integers in the considered interval that are divisible by p^2, p^3, $\ldots$, and divide the value $r[n]$ by p. Thus, if the prime p has a multiplicity of $m \geq 2$ in the prime factorization of an integer n, it is n divisible by p^2, p^3, $\ldots$, p^m, but not by p^{m+1}. So, in this process, we divide the value $r[n]$ exactly $m-1$ times by p, hence after this process $p \mid r[n]$ but $p^2 \nmid r[n]$. When we have done this for all primes p, we have to consider, it is $r[n] = \text{rad}(n)$ for all n in the interval. Thus, Algorithm 1 in fact computes the radicals of all integers in the given interval.

Similarly to the renowned analysis of the sieve of Eratosthenes, it can be concluded that, for each element in the interval on average only a constant number of divisions is required, hence $\mathcal{O}(\text{length})$ steps. Now let $S := \text{start} + \text{length} - 1$. Subsequently, for each prime $p \leq \sqrt{S}$, there are $\mathcal{O}\left(\frac{\log S}{\log p}\right)$ elementary update operations required. That results in $\mathcal{O}(\log S \cdot \log\log S)$ steps in total. Thus, if the set of primes $p \leq \sqrt{S}$ is precomputed, Algorithm 1 uses $\mathcal{O}(\text{length} + \log S \cdot \log\log S)$ time and $\mathcal{O}(\text{length})$ space, since obviously $\text{rad}(n)$ is held in memory for every integer in the interval. If $\text{start} = 1$, hence $\text{length} = S$, this reduces to a time complexity of $\mathcal{O}(S)$ and a space complexity of $\mathcal{O}(S)$.

It is noteworthy that, in the case $\text{start} = 1, \text{length} = S$, a rudimentary implementation of the sieve of Eratosthenes has to sieve with all primes $p \leq S$ to find the set of prime factors for all positive integers up to S, i.e., the radicals of these numbers, and have a time complexity of $\mathcal{O}\left(\sum_{p \leq S} \frac{S}{p}\right) = \mathcal{O}(S \cdot \log S)$. A more efficient variant only sieves with all primes $p \leq \sqrt{S}$, but then must identify the square-free part of each integer in postprocessing. This is a reordering of the steps given in Algorithm 1. However, Algorithm 1 is particularly conducive to high levels of parallelization.

The next insight we observe is that both problems do not ask for the radicals of integers $m < n$ specifically, but only, if the sets $\{\text{rad}(m), \text{rad}(m+1)\}$ and $\{\text{rad}(n), \text{rad}(n+1)\}$ are equal: In the case of $\text{rad}(m) = \text{rad}(n)$ and $\text{rad}(m+1) = \text{rad}(n+1)$, we have found a Benelux pair (m, n); and if $\text{rad}(m) = \text{rad}(n+1)$ and $\text{rad}(m+1) = \text{rad}(n)$, (m, n) is a Benelux pair of second kind. Thus, after computing the radicals for every $n < S$, we build the sets $S_n := \{\text{rad}(n), \text{rad}(n+1)\}$.

A naive way to find duplicates in a list is a pairwise comparison of each two elements of the list. If there are S elements in the list, these are $\mathcal{O}(S^2)$ comparisons. A better and more efficient method is to sort the list and then traverse the sorted list. To do this, we define an easily computable order relation between sets with two elements: Say $\{a_1, a_2\} < \{b_1, b_2\}$ with $a_1 < a_2$ and $b_1 < b_2$, if and only if $a_1 < b_1$ or ($a_1 = b_1$ and $a_2 < b_2$). Then, for two such sets A and B, we clearly always have exactly one of the three possibilities $A < B$, $B < A$ or

```
Input: start and length of an interval to screen
Output: for every n in [start; start + length) the radical r[n] = rad(n)
forall n ∈ [start, start + length) do
    Set r[n] ← n;
end
forall primes p ≤ √(start + length − 1) do
    Set exponent ← 2;
    while p^exponent ≤ start + length − 1 do
        Set res ← start mod p^exponent ;
        if res = 0 then
            Set res ← p^exponent;
        end
        Set shift ← p^exponent − res;
        // So start + shift is the first number in the interval that is
           divisible by p^exponent.
        if shift > length then // nothing more to do for this prime
            continue // Move to next prime.
        end
        forall integers 0 ≤ k ≤ (length−1−shift)/p^exponent do
            Divide r[start + shift + k · p^exponent] by p;
        end
        Increment exponent;
    end
end
return the vector r;
```

Algorithm 1: Computing radicals of all integers in an interval via sieving.

$A = B$. With this order relation, we can sort the sets S_n and find collisions, in other words, integers $m < n$ with $S_m = S_n$, and hence Benelux pairs of first or second kind. Since every Benelux pair of either kind gives rise to a collision, this approach finds every such pair in the given search space. The whole procedure is summarized in Algorithm 2.

Computing all radicals with Algorithm 1 requires $\mathcal{O}(S)$ steps. Building the sets S_n and the list can be achieved in $\mathcal{O}(S)$, too. Sorting this list takes $\mathcal{O}(S \log S)$ steps and the search for duplicates in this sorted list is completed in time $\mathcal{O}(S)$. Combining this gives a time complexity of $\mathcal{O}(S \log S)$, much better than the quadratic naive approach. And since a constant number of arrays of size S are needed, Algorithm 2 has a space complexity of $\mathcal{O}(S)$.

So, if memory is not a limiting factor, the search for Benelux pairs can be done asymptotically as fast as sorting.

```
Input: upper limit S
Output: all Benelux pairs (m, n) of first and second kind with m < n < S
forall 1 ≤ n ≤ S do // Use Algorithm 1 for this loop.
    Compute r[n];
end
forall 1 ≤ n < S do
    Set S_n ← {r[n], r[n + 1]};
    Set list[n] ← (n, S_n);
end
Sort list with respect to (m, S_m) < (n, S_n) ⟺ S_m < S_n;
forall 1 ≤ i < S − 1 do
    Set (m, S_m) ← list[i];
    Set (n, S_n) ← list[i + 1];
    if S_m = S_n then // duplicate found

        if r[m] = r[n] then
            print Benelux pair of 1st kind found: (m, n).
        else
            print Benelux pair of 2nd kind found: (m, n).
        end
    end
end
```

Algorithm 2: Finding all Benelux pairs of first and second kind up to S.

2.2 Breaking the space limitation barrier

Unfortunately, as the capacity for memory is rapidly attained, it becomes a limiting factor. Modern desktop computers utilize CPUs, which possess cache sizes measured in megabytes. As the size of the lists increases, the utilization of the main memory in the RAM becomes imperative, resulting in an approximate tenfold increase in latency. However, the size of such RAM is in the order of gigabytes. Consequently, if the search is to be extended beyond 2^{32}, this approach is inadequate. Upon initial observation, the migration of data to a hard disk appears to be a viable option. However, this approach results in a substantial increase in latency and computing time, with an estimated growth of at least two orders of magnitude. Consequently, it is imperative to devise innovative approaches to engineer an effective search algorithm, even for a more expansive search space.

If we do not have sufficient memory to hold all results of our computation, we have no other choice but to divide the search space into smaller chunks. Let s be the size of such a chunk; in particular, let the chunk C_i be the set of all integers in the interval $[1 + i \cdot (s - 1), 1 + (i + 1) \cdot (s - 1)]$. Here, two successive chunks have to have an overlap of one number since we also need the value of $\mathrm{rad}(n+1)$ for every integer n. This value cannot be computed for the last number in a chunk—so the last number has to be considered at the start of the next chunk a second time.

We can compute the radicals with Algorithm 1 and build the sets S_n for all integers in this chunk in the same way as above in Algorithm 2. We also could find duplicate sets within a chunk in the same way as above. But how can pairs (m, n) with $S_m = S_n$ be found if m and n are in different chunks? For this problem, and in fact for finding duplicates within a chunk, too, we use another approach.

Another efficient way of searching for duplicates is to use hashing: Insert the hash values of the objects under consideration into a hash table and whenever one gets a collision one checks whether the objects in question where equal in the first place. If one uses a hash function and a table size that leads to few collisions for different objects, this should be a fast alternative. This gives us an approach, as used in Algorithm 3, for finding all Benelux pairs (m, n) of both kinds with $m < n$ and n being in the currently examined chunk C_i: Algorithm 3 computes the radical $r[n]$ and set S_n for all numbers n in the given chunk C_i as above. But then it computes hash values h_n for every such set S_n and uses them to insert these data into a previously empty hash table. If there is a collision, the two sets are tested to see if they are equal. If they are, then a Benelux pair has been found. Open addressing can be used to find another slot to insert this data point. Now fix this hash table and recompute the radicals, sets, and hash values for all previous chunks C_j with $j < i$. For every number m in such a previous chunk, search whether h_m is in the above generated hash table. If so, test if the corresponding sets are equal.

If (m, n) is a Benelux pair of either kind with m and n from the same chunk, Algorithm 3 identifies it in Step 9 where the collision is detected while (n, h_n) is being inserted into the hash table. And if m is from a chunk C_j with $j < i$, this pair is found in Step 26 while searching for the hash value h_m of S_m in the hash table. Thus, all Benelux pairs (m, n) with $m < n$ and $n \in C_i$ are found. Since the only values stored in the hash table are hash values h_n from numbers n in the currently investigated chunk C_i, no other potential solutions than the stated ones are reported. Hence, the algorithm is correct.

As the length of each chunk is s, computing the radicals of all numbers in a chunk using Algorithm 1 needs $\mathcal{O}(s + \log S \log\log S)$ steps. Assuming that s is in $\Omega(\log S \log\log S)$ this reduces to $\mathcal{O}(s)$. The sets and hash values of all integers in a given chunk can be computed in $\mathcal{O}(s)$ steps. Let the hash function used be constructed so that two randomly chosen different sets in a family of s such sets have the same hash value only with the probability $p < 1$. If one uses a hash table with size $t \cdot s$, the probability of collisions for different hash values is $\frac{1}{t}$. Thus, while inserting a data point in the hash table in Step 9 of the algorithm, the expected number of slots to be tested until an empty one is found, where the new data can be stored, is $\mathcal{O}\left(\frac{1}{p} \cdot \frac{t}{t-1}\right)$. So, with constants p and t, the hash table can be filled with the data for all numbers in one chunk in $\mathcal{O}(s)$ steps.

Now the loop beginning in Step 18 of Algorithm 3 has to be computed for all $\mathcal{O}\left(\frac{S}{s}\right)$ chunks that came before the current one. In every iteration of this loop, the radicals, sets,

Input: the chunk C_i of integers
Output: all Benelux pairs (m, n) of first and second kind with $m < n$ and $n \in C_i$

Reset the hash table to an empty one;

forall $n \in C_i$ **do** `// Use Algorithm 1 for this loop.`
    Compute $r[n]$;
end

forall $n \in C_i$ **do**
    Set $S_n \leftarrow \{r[n], r[n+1]\}$;
    Compute a hash value h_n from S_n;
    Insert (n, h_n) with key value h_n in the hash table;
    if *there is a collision while* (n, h_n) *is inserted* **then**
        Let (m, h_m) be the date already in the hash table;
        if $S_m = S_n$ **then**
            Test whether (m, n) is a Benelux pair of 1st or 2nd kind and print it;
        end
        Use open addressing to find an unoccupied slot for the current data;
    end
end
`// This finds all Benelux pairs` (m, n) `of both kinds with` $m, n \in C_i$`.`

forall $0 \leq j < i$ **do**
    forall $m \in C_j$ **do** `// Use Algorithm 1 for this loop.`
        Compute $r[m]$;
    end
    forall $m \in C_j$ **do**
        Set $S_m \leftarrow \{r[m], r[m+1]\}$;
        Compute a hash value h_m from S_m;
        Search for the key value h_m in the hash table;
        if *the search is successful* **then**
            Let (n, h_n) be the date in the hash table;
            if $S_m = S_n$ **then**
                Test whether (m, n) is a Benelux pair of 1st or 2nd kind and print it;
            end
            Test until an unoccupied slot is found ; `// Then there can't be another entry with same hash value.`
        end
    end
    `// This finds all Benelux pairs` (m, n) `of both kinds with` $m \in C_j$ `and` $n \in C_i$`.`
end

Algorithm 3: Finding all Benelux pairs (m, n) of both kinds with $m < n$ and $n \in C_i$.

and hash values for all numbers in a chunk C_j can be computed in $\mathcal{O}(s)$ steps, as above. In the same way as in Step 9, searching for a duplicate to a single data point can be done in an expected running time of $\mathcal{O}(1)$, which is $\mathcal{O}(s)$ for all elements of a chunk C_j. Hence, the whole computation for such a chunk C_j needs $\mathcal{O}(s)$ steps and the whole loop beginning in Step 18 has a running time of $\mathcal{O}(S)$. Since $s \leq S$, Algorithm 3 needs $\mathcal{O}(S)$ steps.

But to find all Benelux pairs (m, n) of both kinds with $m < n$, one has to run Algorithm 3 for all $\frac{S}{s-1}$ chunks consisting of integers $\leq S$. Thus, one can identify all Benelux pairs of both kinds in $\mathcal{O}\left(\frac{S^2}{s}\right)$ time. And since at any one time only a constant number of lists of size s is needed, this computation can be achieved while only using $\mathcal{O}(s)$ space.

3 Implementation and results

Already the first steps in implementing Algorithm 2 showed that the memory limitations of the desktop computer available to the author quickly limits the search space. Thus, the most effort was invested in implementing and optimizing Algorithm 3. To get the best possible performance, one should chose the chunk size s to be as large as possible. The author went for 2^{27}. This allowed for a problem-free use of a large part of the memory available. Now the most computationally intensive parts of the algorithm have the advantage of being highly parallelizable. Thus, we optimized the code to use this and utilize the significant parallelism, which is possible on a modern GPU.

3.1 Optimization for parallel computing

A number of optimizations were applied while implementing Algorithm 3. Some decisive ones—in the sense of accelerating the code— are listed below:

- A key part in the algorithm is the computation of the radicals. Given a chunk C_i, one has to identify all multiples of p^k and divide the respective values by p. If the offset, which gives the first number in the chunk that is divisible by p^k, is computed for a given prime power p^k, then these divisions can be done all in parallel since they are independent of each other. Furthermore, if $p < q$ are two primes with $p^4 \geq \text{start} + \text{length}$, then no number in the given chunk can be divisible by p^2 and q^2 at the same time. Thus, no two such primes can influence the computation for the same number and so with all of these primes the sieving process can be done in parallel. If $p^3 \geq \text{start} + \text{length}$, we at least know that no number in the chunk is divisible by p^3 or higher powers, thus we only have to check for multiples of p^2.

- For the prime $p = 2$, we can use a built-in function provided by the programming language and C++ extension CUDA, which we used to write the code for our program utilizing an NVIDIA GPU: ctz, which stands for "count trailing zeros". This function returns the number of zeros that a binary integer ends in, hence $\text{ctz}(n) = x$ is equivalent to $2^x \mid n$, but $2^{x+1} \nmid n$. Thus, for even n, we have to divide n by 2^{x-1} to only let one factor 2 remain; for even n that are not a multiple of 4, there is nothing to do in this context. So, for all n that are multiples of 4, we compute $\mathrm{ctz}(n)$ and then shift n by $\mathrm{ctz}(n) - 1$ bits to the right. Both operations are very fast and can be performed in parallel for all n. Then the prime $p = 2$ is fully processed.

- After the radicals of all numbers in the given chunk are computed, we have to compute the hash values of the respective sets. For this we use a commutative hash function that sends a pair $(\mathrm{rad}(n), \mathrm{rad}(n+1))$ of 64 bit integers to an integer $0 \leq h_n < 2^{29} = 4s$. As this is the size of the hash table, too, the initial hash addresses for two different sets should be equal only in $\frac{1}{4}$ of the cases. Thus, since we use open addressing, on average only a small number of slots has to be examined to find an unoccupied one to insert a new data point. In parallel, we can examine the slot at the initial address, the next one, and so on, writing in shared memory whichever address is the first with an unoccupied slot. In our tests, the best performance was achieved by examine 4 addresses in parallel. The search for duplicates is handled in the same way as the insertion of new data into the hash table. And, of course, the calculation of the hash values, insertions of data into the hash table and searches in the hash table are performed in parallel for all elements of a chunk. One only has to ensure that no two values are inserted at the same time at the same slot in the hash table. This is done with built-in thread-locking mechanisms.

- Since on the GPU only 32 bit integers are supported natively, we had to implement our own 64 bit arithmetic. By doing so we could optimize for our special circumstances. For example, in the recurring task of reducing a 64 bit integer $a_1 \cdot 2^{32} + a_0$ modulo a prime $p < 2^{32}$, we computed $((a_1 \mod p) \cdot (2^{32} \mod p) + (a_0 \mod p)) \mod p$, where the factor $(2^{32} \mod p)$ could be precomputed in advance and only has then to be read from memory.

- Since communication between host memory and device memory—in other words, RAM vs. memory on the GPU—could result in a bottleneck in such a computation, we reduced the communication to a bare minimum: After sending some initial data—such as the list of primes to be used in the sieving process—to the GPU, transfer of data was limited to primarily the chunk number. All other data—such as which numbers are in this chunk and which of them are part of a Benelux pair—are computed entirely on the GPU. Possible findings of Benelux pairs are written into a buffer and sent back to the host system, which writes them into a file.

m	n	$m+1$	$n+1$	Remarks
$75 =$ $3 \cdot 5$	$1215 =$ $3^5 \cdot 5$	$76 =$ $2^2 \cdot 19$	$1216 =$ $2^6 \cdot 19$	exceptional
$2 =$ 2	$8 =$ 2^3	$3 =$ 3	$9 =$ 3^2	$k = 2$
$6 =$ $2 \cdot 3$	$48 =$ $2^3 \cdot 3$	$7 =$ 7	$49 =$ 7^2	$k = 3$
$14 =$ $2 \cdot 7$	$224 =$ $2^5 \cdot 7$	$15 =$ $3 \cdot 5$	$225 =$ $3^2 \cdot 5^2$	$k = 4$
$30 =$ $2 \cdot 3 \cdot 5$	$960 =$ $2^6 \cdot 3 \cdot 5$	$31 =$ 31	$961 =$ 31^2	$k = 5$
$62 =$ $2 \cdot 31$	$3968 =$ $2^7 \cdot 31$	$63 =$ $3^2 \cdot 7$	$3969 =$ $3^4 \cdot 7^2$	$k = 6$
$126 =$ $2 \cdot 3^2 \cdot 7$	$16128 =$ $2^8 \cdot 3^2 \cdot 7$	$127 =$ 127	$16129 =$ 127^2	$k = 7$
$254 =$ $2 \cdot 127$	$65024 =$ $2^9 \cdot 127$	$255 =$ $3 \cdot 5 \cdot 17$	$65025 =$ $3^2 \cdot 5^2 \cdot 17^2$	$k = 8$
$510 =$ $2 \cdot 3 \cdot 5 \cdot 17$	$261120 =$ $2^{10} \cdot 3 \cdot 5 \cdot 17$	$511 =$ $7 \cdot 73$	$261121 =$ $7^2 \cdot 73^2$	$k = 9$
$1022 =$ $2 \cdot 7 \cdot 73$	$1046528 =$ $2^{11} \cdot 7 \cdot 73$	$1023 =$ $3 \cdot 11 \cdot 31$	$1046529 =$ $3^2 \cdot 11^2 \cdot 31^2$	$k = 10$
$2046 =$ $2 \cdot 3 \cdot 11 \cdot 31$	$4190208 =$ $2^{12} \cdot 3 \cdot 11 \cdot 31$	$2047 =$ $23 \cdot 89$	$4190209 =$ $23^2 \cdot 89^2$	$k = 11$
$4094 =$ $2 \cdot 23 \cdot 89$	$16769024 =$ $2^{13} \cdot 23 \cdot 89$	$4095 =$ $3^2 \cdot 5 \cdot 7 \cdot 13$	$16769025 =$ $3^4 \cdot 5^2 \cdot 7^2 \cdot 13^2$	$k = 12$
$8190 =$ $2 \cdot 3^2 \cdot 5 \cdot 7 \cdot 13$	$67092480 =$ $2^{14} \cdot 3^2 \cdot 5 \cdot 7 \cdot 13$	$8191 =$ 8191	$67092481 =$ 8191^2	$k = 13$
$16382 =$ $2 \cdot 8191$	$268402689 =$ $2^{15} \cdot 8191$	$16383 =$ $3 \cdot 43 \cdot 127$	$268402689 =$ $3^2 \cdot 43^2 \cdot 127^2$	$k = 14$
$32766 =$ $2 \cdot 3 \cdot 43 \cdot 127$	$1073676288 =$ $2^{16} \cdot 3 \cdot 43 \cdot 127$	$32767 =$ $7 \cdot 31 \cdot 151$	$1073676289 =$ $7^2 \cdot 31^2 \cdot 151^2$	$k = 15$
$65534 =$ $2 \cdot 7 \cdot 31 \cdot 151$	$4294836224 =$ $2^{17} \cdot 7 \cdot 31 \cdot 151$	$65535 =$ $3 \cdot 5 \cdot 17 \cdot 257$	$4294836225 =$ $3^2 \cdot 5^2 \cdot 17^2 \cdot 257^2$	$k = 16$
$131070 =$ $2 \cdot 3 \cdot 5 \cdot 17 \cdot 257$	$17179607040 =$ $2^{18} \cdot 3 \cdot 5 \cdot 17 \cdot 257$	$131071 =$ 131071	$17179607041 =$ 131071^2	$k = 17$
$262142 =$ $2 \cdot 131071$	$68718952448 =$ $2^{19} \cdot 131071$	$262143 =$ $3^3 \cdot 7 \cdot 19 \cdot 73$	$68718952449 =$ $3^6 \cdot 7^2 \cdot 19^2 \cdot 73^2$	$k = 18$
$524286 =$ $2 \cdot 3^3 \cdot 7 \cdot 19 \cdot 73$	$274876858368 =$ $2^{20} \cdot 3^3 \cdot 7 \cdot 19 \cdot 73$	$524287 =$ 524287	$274876858369 =$ 524287^2	$k = 19$
$1048574 =$ $2 \cdot 524287$	$1099509530624 =$ $2^{21} \cdot 524287$	$1048575 =$ $3 \cdot 5^2 \cdot 11 \cdot 31 \cdot 41$	$1099509530625 =$ $3^2 \cdot 5^4 \cdot 11^2 \cdot 31^2 \cdot 41^2$	$k = 20$

Table 1: All Benelux pairs (m, n) with $m < n < 1.4 \cdot 10^{12}$ and their prime factorizations.

m	$n+1$	$m+1$	n	Remarks
$35 =$ $5 \cdot 7$	$4375 =$ $5^4 \cdot 7$	$36 =$ $2^2 \cdot 3^2$	$4374 =$ $2 \cdot 3^7$	exceptional
$2 =$ 2	$4 =$ 2^2	$3 =$ 3	$3 =$ 3	$k = 0$
$3 =$ 3	$9 =$ 3^2	$4 =$ 2^2	$8 =$ 2^3	$k = 1$
$5 =$ 5	$25 =$ 5^2	$6 =$ $2 \cdot 3$	$24 =$ $2^3 \cdot 3$	$k = 2$
$9 =$ 3^2	$81 =$ 3^4	$10 =$ $2 \cdot 5$	$80 =$ $2^4 \cdot 5$	$k = 3$
$17 =$ 17	$289 =$ 17^2	$18 =$ $2 \cdot 3^2$	$288 =$ $2^5 \cdot 3^2$	$k = 4$
$33 =$ $3 \cdot 11$	$1089 =$ $3^2 \cdot 11^2$	$34 =$ $2 \cdot 17$	$1088 =$ $2^6 \cdot 17$	$k = 5$
$65 =$ $5 \cdot 13$	$4225 =$ $5^2 \cdot 13^2$	$66 =$ $2 \cdot 3 \cdot 11$	$4224 =$ $2^7 \cdot 3 \cdot 11$	$k = 6$
$129 =$ $3 \cdot 43$	$16641 =$ $3^2 \cdot 43^2$	$130 =$ $2 \cdot 5 \cdot 13$	$16640 =$ $2^7 \cdot 5 \cdot 13$	$k = 7$
$257 =$ 257	$66049 =$ 257^2	$258 =$ $2 \cdot 3 \cdot 43$	$66048 =$ $2^8 \cdot 3 \cdot 43$	$k = 8$
$513 =$ $3^3 \cdot 19$	$263169 =$ $3^6 \cdot 19^2$	$514 =$ $2 \cdot 257$	$263168 =$ $2^9 \cdot 257$	$k = 9$
$1025 =$ $5^2 \cdot 41$	$1050625 =$ $5^4 \cdot 41^2$	$1026 =$ $2 \cdot 3^3 \cdot 19$	$1050624 =$ $2^{10} \cdot 3^3 \cdot 19$	$k = 10$
$2049 =$ $3 \cdot 683$	$4198401 =$ $3^2 \cdot 683^2$	$2050 =$ $2 \cdot 5^2 \cdot 41$	$4198400 =$ $2^{11} \cdot 5^2 \cdot 41$	$k = 11$
$4097 =$ $17 \cdot 241$	$16785409 =$ $17^2 \cdot 241$	$4098 =$ $2 \cdot 3 \cdot 683$	$16785408 =$ $2^{12} \cdot 3 \cdot 683$	$k = 12$
$8193 =$ $3 \cdot 2731$	$67125249 =$ $3^2 \cdot 2731^2$	$8194 =$ $2 \cdot 17 \cdot 241$	$67125248 =$ $2^{13} \cdot 17 \cdot 241$	$k = 13$
$16385 =$ $5 \cdot 29 \cdot 113$	$268468225 =$ $5^2 \cdot 29^2 \cdot 113^2$	$16386 =$ $2 \cdot 3 \cdot 2731$	$268468224 =$ $2^{14} \cdot 3 \cdot 2731$	$k = 14$
$32769 =$ $3^2 \cdot 11 \cdot 331$	$1073807361 =$ $3^4 \cdot 11^2 \cdot 331^2$	$32770 =$ $2 \cdot 5 \cdot 29 \cdot 113$	$1073807360 =$ $2^{15} \cdot 5 \cdot 29 \cdot 113$	$k = 15$
$65537 =$ 65537	$4295098369 =$ 65537^2	$65538 =$ $2 \cdot 3^2 \cdot 11 \cdot 331$	$4295098368 =$ $2^{16} \cdot 3^2 \cdot 11 \cdot 331$	$k = 16$
$131073 =$ $3 \cdot 43691$	$17180131329 =$ $3^2 \cdot 43691^2$	$131074 =$ $2 \cdot 65537$	$17180131328 =$ $2^{17} \cdot 65537$	$k = 17$
$262145 =$ $5 \cdot 13 \cdot 37 \cdot 109$	$68720001025 =$ $5^2 \cdot 13^2 \cdot 37^2 \cdot 109^2$	$262146 =$ $2 \cdot 3 \cdot 43691$	$68720001024 =$ $2^{18} \cdot 3 \cdot 43691$	$k = 18$
$524289 =$ $3 \cdot 174763$	$274878955521 =$ $3^2 \cdot 174763^2$	$524290 =$ $2 \cdot 5 \cdot 13 \cdot 37 \cdot 109$	$274878955520 =$ $2^{19} \cdot 5 \cdot 13 \cdot 37 \cdot 109$	$k = 19$
$1048577 =$ $17 \cdot 61681$	$1099513724929 =$ $17^2 \cdot 61681^2$	$1048578 =$ $2 \cdot 3 \cdot 174763$	$1099513724928 =$ $2^{20} \cdot 3 \cdot 174763$	$k = 20$

Table 2: All Benelux pairs (m, n) of second kind with $m < n < 1.4 \cdot 10^{12}$ and their prime factorizations.

3.2 List of findings

Using the optimized implementation of Algorithm 3, we were able to extend the previously considered search space by a large factor: In approximately one minute of computing time on our standard desktop PC with a consumer graphics card from 2021, we reached $S = 2^{32}$, and thus the upper limit of previous searches. Since the algorithm for fixed s is quadratic in the upper search limit S, it took approximately one and a half months to reach $S = 2^{40} \approx 1.0995 \cdot 10^{12}$. In fact, we searched a little further. From this we can conclude:

Theorem 1. *Let $m < n < 1.4 \cdot 10^{12}$ be positive integers. Then*

a) (m, n) is a Benelux pair (of first kind) if and only if

- *$m = 75$ and $n = 1215$ or*
- *$m = 2^k - 2$ and $n = 2^{2k} - 2^{k+1}$ for an integer $2 \leq k \leq 20$ and*

b) (m, n) is a Benelux pair of second kind if and only if

- *$m = 35$ and $n = 4374$ or*
- *$m = 2^k + 1$ and $n = 2^{2k} + 2^{k+1}$ for an integer $0 \leq k \leq 20$.*

Table 1 lists all found Benelux pairs with their prime factorizations and Table 2 lists all found Belenux pairs of second kind.

References

[1] T. F. Bloom, Erdős Problem #850, 2025, https://www.erdosproblems.com/850.

[2] P. Erdős, Some problems I presented or planned to present in my short talk, in B. C. Berndt, H. G. Diamond, and A. J. Hildebrand, eds., *Analytic Number Theory—Proceedings of a Conference in Honor of Heini Halberstam*, Vol. 1, Birkhäuser, 1996, pp. 333–335.

[3] R. K. Guy, *Unsolved Problems in Number Theory*, 3rd edition, Springer, 2004.

[4] Third Benelux Mathematical Olympiad, Problem 1, 2011, http://www.bxmo.org/problems/bxmo-problems-2011-zz.pdf.

2020 *Mathematics Subject Classification*: Primary 11-04; Secondary 11Y55.
Keywords: Erdős problem, Benelux pair, integers with same prime factors.

(Concerned with sequences A088966 and A343101.)

Received June 1 2025; revised versions received July 20 2025; December 7 2025; December 12 2025. Published in *Journal of Integer Sequences*, December 12 2025.

Return to Journal of Integer Sequences home page.