3D integer factorization

João Carlos Leandro da Silva

Via Medole 22, Castiglione delle Stiviere (MN), Italy

We propose a new view for integer factorization based on a radical design. In the traditional model, the computer experiment consists of physical devices (hardware) which in turn contain programs (software) that run an algorithm equivalent to some mathematical theory. In the present model, the mathematical theory corresponds to a finite set of basic algorithms distributed among the hardware and all of these constitute the computer experiment itself. All of the currently known factoring algorithms consist of several steps in order to find any non-trivial factor. Such algorithms may run on a single or many machines. Our method is a single step – the generation of a multiple of any prime factor. Consequently, factoring reduces to searching efficiently for one such multiple. If the given modulus has m decimal digits then m^2 machines or physical cores are required. The resulting search experiment is a square matrix of $m\ x\ m$ computers. This method is deterministic and revolves around finding a three-dimensional point (x, y, z) that will lead to a successful factorization.

1. INTRODUCTION

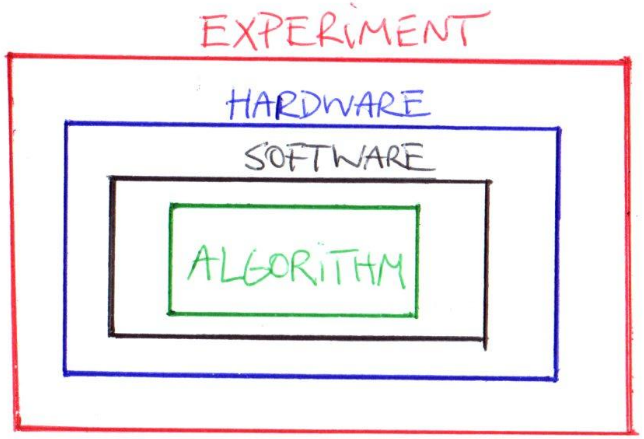
The fundamental theorem of arithmetic states that every integer greater than 1 is either a prime or a finite product of prime numbers and such product is unique. For example, 28 is four times seven or 28 = 2 × 2 × 7 and such is the only way you can express the integer 28 in terms of prime numbers. In fact, the unique-prime-factorization theorem is another name for the fundamental theorem of arithmetic [1]. Finding ways to break or factor composite integers into their respective primes goes back a long time but with the birth of public-key cryptography [2], the interest has literally exploded [3]. Today, computer scientists refer to these methods as integer factorization algorithms [4]. Among the most used algorithms is the Elliptic Curve Method (ECM), the Quadratic Sieve (QS) and the Number Field Sieve (NFS). Although such techniques have achieved remarkable factorization records during the past decades, they all suffer from several drawbacks. First, their dependence on smooth numbers or random integers. Second, their inability to scale-up due to a bottleneck effect in at least one of their steps. Third, all these algorithms date back to 1995, that is, in the past twenty-four years nobody has invented a new factorization method [5]. In this work, we provide an alternative path to the status quo. 3D integer factorization does not use smooth numbers. It has a single step so as the modulus increases either it works (outputs a non-trivial factor) or simply keeps on running. Further, this novel approach introduces original ideas and concepts that are worth considering.

2. NEW MODEL

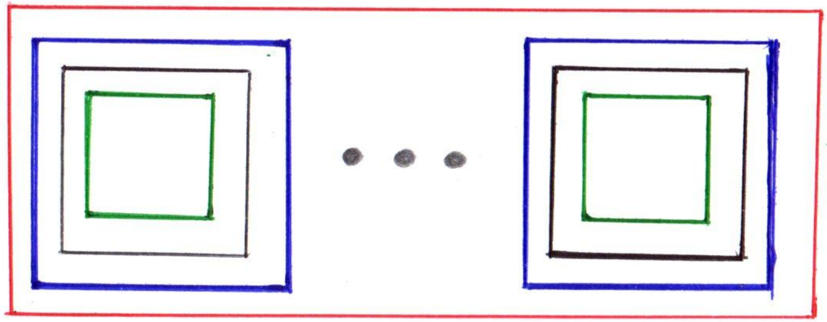
In computer science, researchers frequently use terms such as experiment, hardware, software and algorithm [6]. What do these words really mean? Curious is the fact that no formal definition for algorithm exists. However, there is a general agreement regarding its properties. First, an algorithm has a finite description. Second, it is composed of “basic” steps. Third, the amount of resources required by the algorithm at any given time must be finite. Fourth, the next step depends on the result of the previous step. Fundamentally, an algorithm is a finite procedure that we will implement on a machine via a programming language [7]. Following, we suggest that the notions of algorithm and computer experiment are much more ample than expected. As shown in page 4, in the conventional model, an experiment comprises hardware of different types such as monitors and printers. These, in turn, contain software that run many algorithms. It is clear that all this can be associated with a single machine. For example, suppose that our experiment consists in recording the number of vehicles and the respective license plates of the cars parked in front of your house from 7 am to 7 pm. In this case, the hardware is a laptop and a smartphone.

The software is *Microsoft Office* and the algorithm is an Excel file. If more than one computer is used then the experiment incorporates all hardware, all software and every algorithm needed to carry it out. Such depicts the case of multiple machines as displayed on the left side of page 4. Therefore, in the conventional model there is no significant difference between an experiment for a single and that for multiple machines. On the other hand, in the new model major alterations occur. A 10 by 10 matrix enclosing 100 squares is on the right side of page 4. Each square contains a blue capital letter M that stands for machine with a number subscript ranging from 1 to 100 symbolizing the software count. The given matrix has an exterior red contour and an internal green grid. What we have just described is the new model. It represents a new method to factor a 10-digit modulus like 1476602513. In theory, since the modulus has 10 decimal digits then we need 100 computers. Each machine or hardware will occupy a square of the green grid. In addition, each machine contains a numbered copy of a software called *Mathematica* denoted by the software count. Within such proprietary program a basic algorithm exists that joined with all of its 100 copies constitutes the experiment itself as defined by the red contour. Note that in the new model there is no such thing as a factoring algorithm running on a single or multiple machines. Instead, we have a search experiment composed of 100 machines each containing a basic algorithm with a unique three-dimensional point (x, y, z). Thus, in the new model (what was once understood as the "algorithm") now has been split or partitioned into 100 different basic algorithms as shown by the internal green grid on page 4. In general, given a modulus with m decimal digits then m^2 machines are required. Yet, when the modulus is small like 1476602513 it is possible to use a single computer. The previous description shows that experiment, hardware, software and algorithm are not four distinct layers of computer science. We have revealed with respect to experiment and algorithm, a new and interesting connection that will aid in the design of innovative integer factorization methods.

CONVENTIONAL MODEL



SINGLE MACHINE



MULTIPLE MACHINES

NEW MODEL

M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}
M_{11}	M_{12}	M_{13}	M_{14}	M_{15}	M_{16}	M_{17}	M_{18}	M_{19}	M_{20}
M_{21}	M_{22}	M_{23}	M_{24}	M_{25}	M_{26}	M_{27}	M_{28}	M_{29}	M_{30}
M_{31}	M_{32}	M_{33}	M_{34}	M_{35}	M_{36}	M_{37}	M_{38}	M_{39}	M_{40}
M_{41}	M_{42}	M_{43}	M_{44}	M_{45}	M_{46}	M_{47}	M_{48}	M_{49}	M_{50}
M_{51}	M_{52}	M_{53}	M_{54}	M_{55}	M_{56}	M_{57}	M_{58}	M_{59}	M_{60}
M_{61}	M_{62}	M_{63}	M_{64}	M_{65}	M_{66}	M_{67}	M_{68}	M_{69}	M_{70}
M_{71}	M_{72}	M_{73}	M_{74}	M_{75}	M_{76}	M_{77}	M_{78}	M_{79}	M_{80}
M_{81}	M_{82}	M_{83}	M_{84}	M_{85}	M_{86}	M_{87}	M_{88}	M_{89}	M_{90}
M_{91}	M_{92}	M_{93}	M_{94}	M_{95}	M_{96}	M_{97}	M_{98}	M_{99}	M_{100}

MULTIPLE MACHINES

3. FACTORING ALGORITHM = SEARCH EXPERIMENT

Our analysis only considers classical computers (Turing machines) where each computer has a finite amount of random access memory (RAM) and a central processing unit (CPU) of finite speed. We deal with deterministic algorithms, that is, if an input X results in output Y then every time we run that algorithm with input X the output will always be Y. Assume the modulus is a semiprime or a composite of two different but equally sized prime factors. What are the main similarities among the known integer factorization algorithms? First, all three algorithms (ECM, QS and NFS) require some type of pre-computation as first step. Second, they all consist of a finite sequence of intermediate steps but not every such step is polynomial with respect to the input. Third, their final step is always the evaluation of the greatest common divisor between some specific numerical calculation and the given modulus. If the result is a non-trivial factor then we have a successful factorization. Please realize that such can only occur if the specific numerical calculation is a multiple of either prime factor. At this stage, a natural question comes to mind. The final aim of any integer factorization algorithm is to find a prime factor as swift as possible. Then, why not try to invent new algorithms based on the generation of a multiple of any prime factor? Regretfully, it seems that very few people believe that such is worth pursuing.

To put it bluntly, factoring boils down to searching and efficient factoring is finding an express path to perform a successful search. From here on, we will substitute the commonly used expression “factoring algorithm” with “search experiment” or SE for short. Since the modulus is a semiprime, the mathematical theory states that there is an infinite number of multiples regarding either prime factor. Our aim is to find one such multiple as fast as possible. At the core of the SE there is a basic algorithm. It consists of a seed, a generator and a key. The seed changes the modulus into another integer. The generator has three inputs. The first input is the seed or an integer in base 10. The second input consists of a sequence of consecutive positive integers. The generator takes the seed and converts it to many different bases (as many as the sequence) producing as output an array of integers. The key represents a three-dimensional point (x, y, z) where all three coordinates are positive integers. If the given modulus has m decimal digits then m^2 computers are required. The finite set of algorithms are m^2 copies of the basic algorithm distributed among m^2 machines. Yet, the values for both x and y in each of these copies are distinct and constant. The third input of the generator is the variable z. Therefore, the search experiment consists in running simultaneously all m^2 computers where z starts at one and is incremented at every iteration. Please note that the variable z will determine if a multiple of any prime factor exists. In such case, the specific computer will halt and the respective algorithm will output any non-trivial factor of the given modulus. The search is complete and the factorization achieved. In several ways, 3D integer factorization is related to 3D cryptography [8]. The first views the modulus not as a mere composite number but as a mathematical cipher, that is, an integer that hides through the elementary operation of multiplication two unknown prime numbers. In order to break such cipher, our method does not use the power of mathematics alone but brings along the principles of basic cryptanalysis. In fact, the search experiment via its square matrix of $m\, x\, m$ computers is just another architecture built to test simultaneously as many possible combinations of (x, y, z) points as we can. Our hope is that at least one of these machines will find a key that will open the hidden door to a valid factorization in a reasonable span of time. We will begin to explain everything in detail. What exactly indicates a multiple of any prime factor? Two illustrative examples will clear any doubts. Suppose our modulus is 15 where 3 and 5 are its prime factors. Then, any multiple of 3 and any multiple of 5 will do the job in a single step because GCD[15, 6] = 3 and GCD[15, 10] = 5 where GCD stands for the greatest common divisor. Likewise, if the modulus is 143 where 11 and 13 are its prime factors, then any multiple of 11 and 13 will result in a successful factorization since GCD[143, 22] = 11 and GCD[143, 26] = 13 as expected. We will demonstrate that there are many ways to generate such multiples.

4. GENERATION OF A MULTIPLE OF ANY PRIME FACTOR

Provided all the decimal digits of the modulus, what can we do with only such piece of information? Surprisingly, a great deal! Remember the previous examples, 15 and 143. Pick 15, however, now regard this integer as a list or {1, 5} to be exact. Given a list of k distinct objects such list can be organized in $k!$ arrangements. Since both digits in our list are distinct, the number of permutations is 2! = 2. In fact, the permutations are {1, 5} and {5, 1} but after joining the respective digits inside each list, the result is 15 and 51, respectively. It is easy to see that 51 is a multiple of 3, hence GCD[15, 51] = 3 and we have found a way to generate a multiple of one of the prime factors of 15. Repeating this process again with 143, the number of permutations is 3! = 6 or {143, 134, 413, 431, 314, 341}. Only 341 is a multiple of 11, consequently, GCD[143, 341] = 11 as expected. At last, we select the integer 2923 or the product of 37 and 79. In this case, the number of permutations is not twenty-four (4!) but twelve. The reason for such lies in the undeniable observation that the digit 2 is repeated twice within the list {2, 9, 2, 3}. Hence, if the modulus has k decimal digits and some digits occur more than once, then the total number of permutations is smaller than $k!$ but bigger than one. With respect to the previously mentioned twelve permutations, only two numbers (9322 and 3922) are multiples of 79 and 37, respectively. If we now consider 175337 or the product of 271 and 647, there are 180 permutations but only one of these is a multiple of 271. Actually, GCD[175337, 775331] = 271. Pick 32927347 there are 5,040 permutations, but only two of these are multiples of the respective prime factors. Thus, GCD[32927347, 33774922] = 3767 and GCD[32927347, 92243773] = 8741. Finally, elect 529356695359 as the last example. Even though there are 831,600 permutations, none of these is a multiple. Here, we fail to factor the given modulus. In conclusion, as the modulus increases the number of permutations grows very fast making it unfeasible from a computational stand. What happens if we pick a permutation at random? Such will not work either because there is no mathematical proof that among all the permutations there is a multiple.

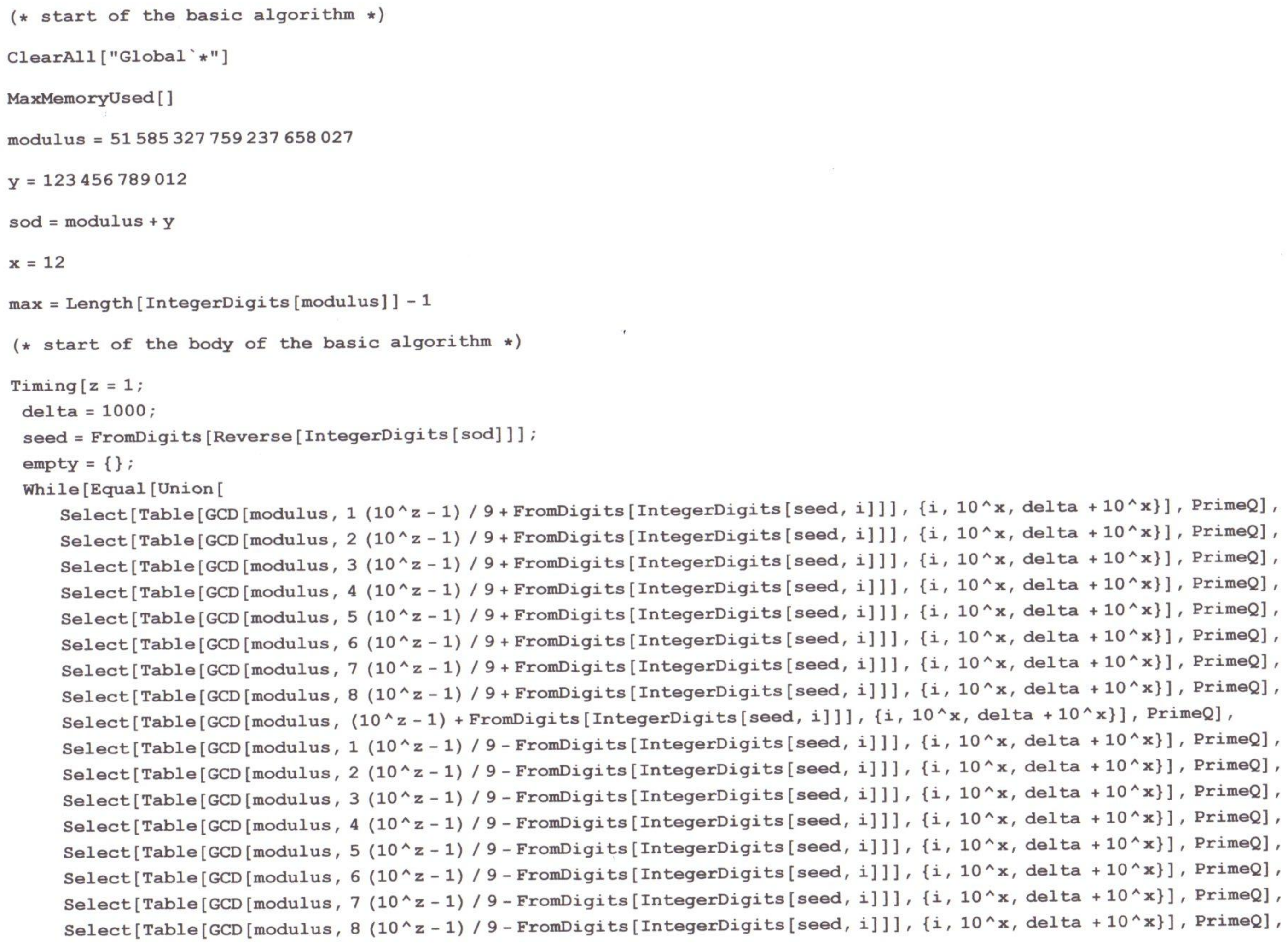
We make another attack but this time with repunits. Repunits are integers composed of a single decimal digit but repeated many times like 1111111. Their mathematical formula is $(10^r - 1)/9$ where r stands for the number of digits. For example, the previous repunit has seven digits so in order to generate it substitute $r = 7$ into the above formula. In my opinion, repunits can factor any composite number. In fact, GCD[2923, 111] = 37 where $r = 3$ and GCD[175337, 11111] = 271 where $r = 5$. The next modulus is 32927347 and GCD[32927347, repunit] = 3767 where $r = 3766$. Remember that we failed to factor 529356695359 using permutations. Now, we get GCD[529356695359, repunit] = 744431 where $r = 74443$ and GCD[529356695359, repunit] = 711089 where $r = 88886$. Does this mean that repunits are more efficient than permutations? The answer is no since as the modulus increases so does the size of r making it impractical. For example, trying to factor a number like RSA-1024 bits (309 decimal digits) which in theory should be the product of a 155 decimal prime and a 154 decimal prime is literally impossible because the size of r must be at least 155 decimal digits. Yet, repunits possess an interesting property. The infinite sequence of ones is both a decimal number and a binary number. For instance, in our previous example GCD[2923, 111] = 37 the repunit 111 is the product of 3 and 37. The built-in function **GCD** of *Mathematica* from Wolfram Research Inc. considers all the arguments of such function as decimal numbers by default. On the other hand, if 111 is viewed as a binary number then it would be equivalent to 7 but GCD[2923, 7] = 1 and that is not our objective. Anyway, is there an integer in another base other than ten that is a multiple of any prime factor of 2923? Yes and there are many. For example, 2923 in base 27 is 407 and GCD[2923, 407] = 37. Likewise, 2923 in base 42 is 395 and GCD[2923, 395] = 79 as expected. Once again, we use another built-in function **IntegerDigits** to obtain both 407 and 395. Also, 175337 in base 99 and 32927347 in base 3777 give successful factorizations. Still, this variation will not work either since it is evident that as the modulus increases so does the numerical value of the corresponding base.

Another important aspect when trying to generate a multiple of any prime factor is the size or the number of decimal digits of the respective multiple. In the previous examples with repunits, the size of r is always greater than the number of decimal digits of the smallest prime factor. For example, the modulus 175337 is the product of 271 and 647. Both prime factors have three decimal digits and GCD[175337, 11111] = 271 where $r = 5$. It is obvious that 111 or $r = 3$ could never factor 175337 since 111 is smaller than both 271 and 647. Therefore, given a semiprime or modulus with m decimal digits, its smallest prime factor must have at least $\frac{m}{2}$ decimal digits. If m is even then it is straightforward but if m is odd, divide it by two and take the integer value as the result. Hence, any potential multiple must have at least $\frac{m}{2}$ decimal digits. For example, RSA-1024 bits is a semiprime with 309 decimal digits. If we divide 309 by two and remain with the integer value, 154 is the result. Accordingly, if you ever try to design a generator of multiples for RSA-1024 bits make sure that any output of the respective generator is at least 154 decimal digits in size, otherwise, your task is unattainable.

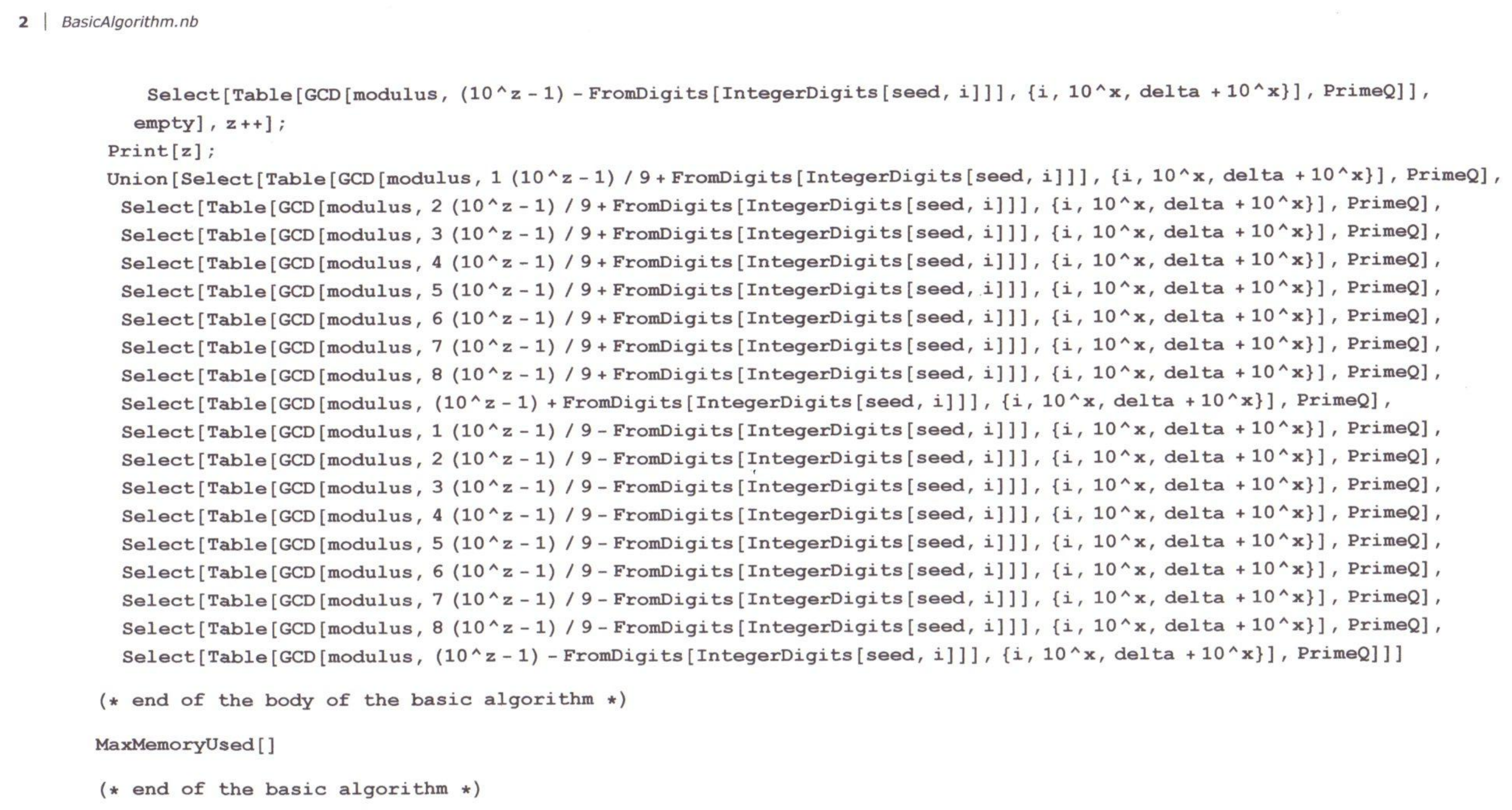
5. BASIC ALGORITHM

We learned that permutations and repunits by themselves do not lead to an efficient mechanism for generating multiples. The way to go is to combine a class of repunits with a collection of different bases but of a higher power. What do we mean? The formula $1(10^r - 1)/9$ creates the sequence of ones while $2(10^r - 1)/9$ produces the sequence of twos and so on. It is clear that the general formula is $k(10^r - 1)/9$ where k ranges from one to nine so the class of repunits is the corresponding nine infinite sequences. Remember **IntegerDigits** we will use it again to present our argument. Base 99 successfully factored 175337 but such required calling ninety-eight times the above built-in function. Another option is to pick a small interval and increase it by orders of ten. For example, setting the interval to nine calls the function only five times. Because **IntegerDigits** starts at $[10^1, 9 + 10^1]$ and ends at $[10^5, 9 + 10^5]$ so base 100009 factors 175337. This is what we mean by a collection (one for each repunit sequence) of different bases but of a higher power.

Our basic algorithm is a *Mathematica* notebook (see pages 12-13) and the size of the code is very small. The first line of *Mathematica* code clears all internal variables. The second line calculates the amount of memory used at the start of the notebook. The third line defines the number to be factored or $modulus$. The fourth line sets y or the second coordinate of the key. Such variable is a cyclic integer (1 followed by 12 followed by 123 and so on), that is, if we use a single computer then after eleven runs $y = 12345678901$ because following each run, the value of y must be changed manually. The fifth line identifies the variable sod as the sum or difference between the given $modulus$ and y. Such variable is the input of the seed. The sixth line is x or the first coordinate of the key. Recall **IntegerDigits** and the respective range that starts at $[10^1, 9 + 10^1]$ and ends at $[10^5, 9 + 10^5]$. There are two distinct powers of ten (1 and 5), that is, $x = 1$ and $x = 5$, respectively. The seventh line refers to the variable max which is the maximum value of the first coordinate of the key or x. Next, we introduce the body of our basic algorithm which consists of a single *While* loop. It is contained within a built-in function **Timing** which calculates the running time in seconds of all the operations inside the body. A few variables and constants are initialized outside of the loop. The third coordinate of the key is set to $z = 1$ where z corresponds to the variable r in the formula $k(10^r - 1)/9$ for the repunits. Thus, z determines not only the size of all repunits but also if a multiple of any prime factor of the given modulus exists. In fact, if such is not the case then z will go on (keep increasing by increments of one) until the full resources (RAM, HDD, etc) of the computer are used. The constant $delta$ refers to the interval (in our previous example it was 9) but here it is set to 1000. Since we are dealing with a search experiment (SE) it is important to introduce two new terms: search space and size of the search space. Suppose that an oracle asks you to pick any prime number of any size. Further, the oracle tells you that it will search inside your mind and find the chosen prime. Such oracle must be godlike or incredibly lucky since the search space is $[2, \infty)$ and the size of the search space is infinity. Ideally, the size of the search space (SSS) is equal to the search space (SS). In the real world, SSS << SS. In the basic algorithm $delta = SSS$ and the selection of its numerical value requires great caution. If $delta$ is too big, the SE will run very slowly and if it is too small, the SE will be fast but may not find any prime factor. Depending on the physical characteristics of the SE itself, the balance point is determined. For example, in our basic algorithm $delta = 1000$ because SE is a single machine (Intel Core i7-6700 CPU @ 3.4 GHz) with 16 GB of RAM. In other words, such numerical value for delta is the right balance between size and speed with respect to the computer at hand. The next constant is $seed$ which takes as input the variable sod and reverses its decimal digits. The final constant is $empty$ and represents an empty set or simply the fact that no prime factor exists. Ultimately, we explain the inner workings of the *While* loop. Essentially, the loop is the generator. It amounts to a mix of repunits and the seed represented in different bases. The mix consists in adding and subtracting both repunits and bases. At every iteration of the *While* loop the following happens. First, the $seed$ is converted into 1000 different integers. Second, these 1000 positive integers get added and subtracted to nine different sequences of repunits. Third, the resulting 1000 integers are all potential multiples. Yet, if no multiple of any of the prime factors is generated then z or the third coordinate of the key is incremented by one and the loop runs again. On the other hand, if a multiple exists, the loop terminates and outputs z and any respective prime factor. In addition, the loop also outputs the total time (in seconds) required to factor the given modulus. The last line of code of the basic algorithm calculates the maximum amount of memory used at the end of the notebook titled **BasicAlgorithm.nb** . The next two pages regard only the code of the basic algorithm while the pages 14-15 relate to its evaluation displayed by **In[ ]:=** for input and **Out[ ]:=** for output. Note that comments start with **(*** and end with ***)** as shown in the next two pages.

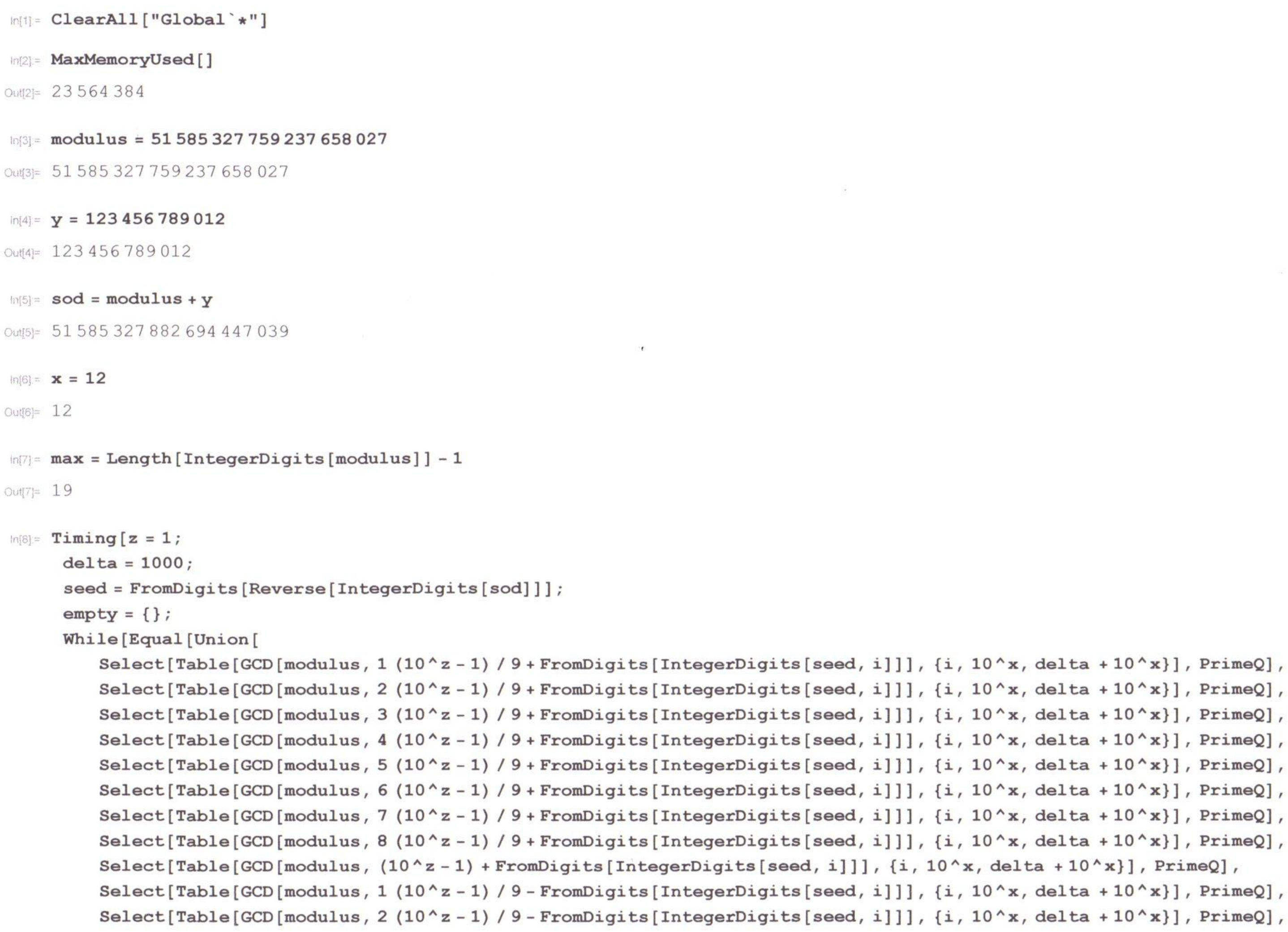


```
(* start of the basic algorithm *)

ClearAll["Global`*"]

MaxMemoryUsed[]

modulus = 51 585 327 759 237 658 027

y = 123 456 789 012

sod = modulus + y

x = 12

max = Length[IntegerDigits[modulus]] - 1

(* start of the body of the basic algorithm *)

Timing[z = 1;
 delta = 1000;
 seed = FromDigits[Reverse[IntegerDigits[sod]]];
 empty = {};
 While[Equal[Union[
    Select[Table[GCD[modulus, 1 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 2 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 3 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 4 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 5 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 6 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 7 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 8 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, (10^z - 1) + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 1 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 2 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 3 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 4 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 5 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 6 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 7 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
    Select[Table[GCD[modulus, 8 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
```

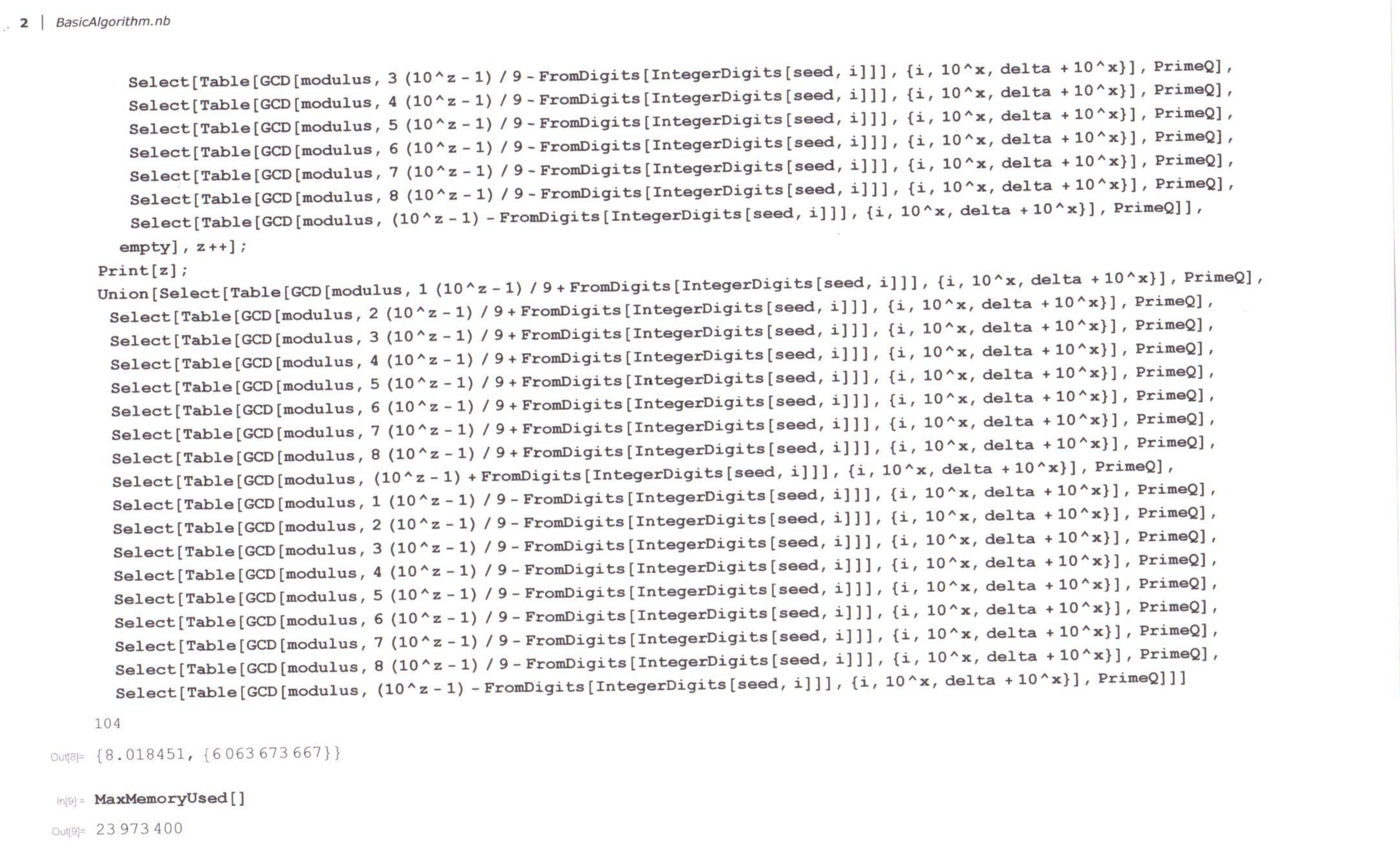


2 | *BasicAlgorithm.nb*

```
    Select[Table[GCD[modulus, (10^z - 1) - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ]],
   empty], z++];
 Print[z];
 Union[Select[Table[GCD[modulus, 1 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 2 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 3 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 4 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 5 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 6 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 7 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 8 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, (10^z - 1) + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 1 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 2 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 3 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 4 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 5 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 6 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 7 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, 8 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
  Select[Table[GCD[modulus, (10^z - 1) - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ]]

(* end of the body of the basic algorithm *)

MaxMemoryUsed[]

(* end of the basic algorithm *)
```



```
In[1]:= ClearAll["Global`*"]

In[2]:= MaxMemoryUsed[]

Out[2]= 23 564 384

In[3]:= modulus = 51 585 327 759 237 658 027

Out[3]= 51 585 327 759 237 658 027

In[4]:= y = 123 456 789 012

Out[4]= 123 456 789 012

In[5]:= sod = modulus + y

Out[5]= 51 585 327 882 694 447 039

In[6]:= x = 12

Out[6]= 12

In[7]:= max = Length[IntegerDigits[modulus]] - 1

Out[7]= 19

In[8]:= Timing[z = 1;
         delta = 1000;
         seed = FromDigits[Reverse[IntegerDigits[sod]]];
         empty = {};
         While[Equal[Union[
             Select[Table[GCD[modulus, 1 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
             Select[Table[GCD[modulus, 2 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
             Select[Table[GCD[modulus, 3 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
             Select[Table[GCD[modulus, 4 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
             Select[Table[GCD[modulus, 5 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
             Select[Table[GCD[modulus, 6 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
             Select[Table[GCD[modulus, 7 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
             Select[Table[GCD[modulus, 8 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
             Select[Table[GCD[modulus, (10^z - 1) + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
             Select[Table[GCD[modulus, 1 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
             Select[Table[GCD[modulus, 2 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
```



```
      Select[Table[GCD[modulus, 3 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
      Select[Table[GCD[modulus, 4 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
      Select[Table[GCD[modulus, 5 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
      Select[Table[GCD[modulus, 6 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
      Select[Table[GCD[modulus, 7 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
      Select[Table[GCD[modulus, 8 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
      Select[Table[GCD[modulus, (10^z - 1) - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ]],
    empty], z++];
  Print[z];
  Union[Select[Table[GCD[modulus, 1 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 2 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 3 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 4 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 5 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 6 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 7 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 8 (10^z - 1) / 9 + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, (10^z - 1) + FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 1 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 2 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 3 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 4 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 5 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 6 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 7 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, 8 (10^z - 1) / 9 - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ],
   Select[Table[GCD[modulus, (10^z - 1) - FromDigits[IntegerDigits[seed, i]]], {i, 10^x, delta + 10^x}], PrimeQ]]]

104

Out[8]= {8.018451, {6 063 673 667}}

In[9]:= MaxMemoryUsed[]

Out[9]= 23 973 400
```

6. SEARCH EXPERIMENTS AND RESULTS

In the last two pages, the previous experiment becomes search experiment nine or SE9, for short, as shown in the table below:

SE#	**SS**	**SSS**	**x**	**y**	**z**	**size of modulus (decimal digits)**	**prime factor(s)**	**time to factor (seconds)**
1	$[10^1, 10^3]$	1	1	0	1	4	37 and 79	0
2	$[10^1, 10^5]$	10	1	0	3	6	271	0
3	$[10^2, 10^7]$	100	2	0	2	8	3767	0
4	$[10^2, 10^9]$	100	2	0	4	10	18211	0
5	$[10^4, 10^{11}]$	1000	4	0	15	12	711089	1
6	$[10^4, 10^{13}]$	1000	4	123456789	32	14	5894327	2
7	$[10^4, 10^{15}]$	1000	9	12	34	16	65774893	2
8	$[10^4, 10^{16}]$	1000	13	0	32	17	116092003	2
9	$[10^4, 10^{19}]$	1000	12	123456789012	104	20	6063673667	8

It is very tempting to look above and imagine all sorts of patterns and relationships so before you jump into any conclusion, several remarks are of utmost importance. First, each search experiment is associated to a single modulus. Second, it is self-evident that as the modulus increases so does the search space (SS) and the same appears to occur with respect to the size of the search space (SSS) but it is not so. 3D integer factorization should scale-up to a composite like RSA-1024 bits still using SSS = 1000. Third, the range of the first coordinate of the key or x is always small. For example, even for the RSA semiprime, the range is between 4 and 308 inclusive. Fourth, recall that the second coordinate of the key or y is a cyclic integer so only 309 cycles are necessary for RSA-1024 bits. Fifth, it seems that as the modulus increases so does the third coordinate of the key or z but such is misleading. In theory, for any given search experiment, there are many different (x, y, z) points. Each value of z reported above represents the first instance of a successful factorization. Hence, there may be other numerical values (smaller or larger) for z that result in a non-trivial factor. Sixth, the times (in seconds) recorded above regard nine computer simulations and not the actual running of the respective search experiments. To be more precise, SE1 pertains to 2923; SE2 to 175337;

SE3 to 32927347; SE4 to 1476602513; SE5 to a 12-digit semiprime or 529356695359; SE6 to 54090759402923; SE7 to a 16-digit modulus or 6427204259252773; SE8 to 50279637935012947 and finally SE9 to 51585327759237658027. Unfortunately, throughout this research, we only had a single computer. Thus, computer simulation simply means using the same machine many times instead of multiple machines once. SE9 concerns a semiprime with 20 decimal digits so 400 physical cores are needed but only one computer is available, so we manually changed the settings within the basic algorithm many times. How? Open the notebook **BasicAlgorithm.nb** and set $y = 0$ and $x = 4$ then run it for 10 seconds. If during this period there is no output, abort the evaluation of the notebook. Next, set $y = 0$ and $x = 5$ repeat the same actions until $y = 0$ and $x = 19$. Continue this process but with $y = 1$ and $x = 4$ followed by $y = 1$ and $x = 5$ until $y = 1$ and $x = 19$. It is important to note that next $y = 12$ and $x = 4$ followed by $y = 12$ and $x = 5$ until $y = 12$ and $x = 19$. After two hundred and one evaluations of the above notebook, we discover that for $y = 123456789012$ and $x = 12$ with $z = 104$ a prime factor is obtained in 8 seconds. Please note that such is equivalent to a search experiment of 201 computers with 201 different (x, y, z) points running for 10 seconds. Now, we can justify why the previous times (in seconds) may not be the minimum factorization time for each given search experiment. In SE9, we needed 400 computers to test 400 different (x, y, z) points so 199 distinct (x, y, z) points are missing. This means that among these there could be some (x, y, z) point where the value of z is smaller than 104 and, in such case, the factorization time would be less than 8 seconds. In fact, such is the reason behind 3D integer factorization requiring m^2 computers running simultaneously for a modulus with m decimal digits. All machines must run concurrently so that the m^2 different (x, y, z) points are tested. In fact, it is impossible to know beforehand which machine will find a prime factor first. A last remark on SE9, in **BasicAlgorithm.nb** look at the numerical values corresponding to **Out[9]** and **Out[2]**. Their respective difference (23,973,400 – 23,564,384) is the actual amount of memory (409,016 bytes) needed to find the prime in the previous table.

We just mentioned that all m^2 computers must run at the same time in order to search for a valid key. Yet, there is also the possibility that all computers keep running or never halt because the required value for z is too big. In my opinion, two are the ways to validate this new scheme. One, repeat SE1 thru SE9 but this turn using the respective required machines and for each SE record the minimum factoring time. Continue this process with higher moduli (for instance, until $m = 50$) and plot the results in a two-dimensional graph where the horizontal axes stands for the size of the modulus in decimal digits and the vertical axes for the factorization time in seconds. The resulting graph should be a straight line. Calculate its slope to determine the validity of 3D integer factorization. If the slope is big, the method fails to scale-up. On the other hand, if it is zero or small then repeat the entire process with even higher moduli ($m = 100$) and graph again to check if the slope did not vary. If it is still small then it is very likely that this innovative technique is efficient. Yet, before you start, recall that for a modulus with 100 decimal digits 10000 computers are required. Such large quantity of machines is difficult to find but they exist in the form of distributed computing or as supercomputers. Currently, the largest known supercomputer is Sunway TaihuLight and is located in China. It has more than ten million physical cores which means that even RSA-2048 bits (617 decimal digits) could be tested using the present method. Two, we sincerely hope that someone will consider this paper interesting and as a result will try to formalize the current method. If this happens, one would be able to mathematically prove or disprove the validity of 3D integer factorization. In other words, if a mathematician or computer scientist can formalize this work then, in theory, it would be possible to understand the bounds (lower and upper) with respect to the size of the modulus without doing any experiment. Regardless, please acknowledge that if this approach is not suitable for larger moduli, perhaps a variation of it and/or an adequate modification either mathematical or conceptual may do the task. Our objective is to find either alone or in collaboration an efficient method or technique that will definitely solve the integer factorization problem in the positive.

7. CONCLUSIONS

We believe that the design principles of simplicity and symmetry should be the pillars for the next generation of factoring algorithms. Simple means many things. First, the algorithm is deterministic and uses a small portion of the total RAM available within the given computer system independent of the modulus. Second, the algorithm has few steps and the size of the code is small. Third, the body of the algorithm contains two loops (at most) and we can measure its running time in seconds with precision. Fourth, all built-in functions are polynomial with respect to the input. Every mathematical operation employed runs in polynomial time. Likewise, symmetrical can be used to characterize several things. First, it is evident that the mathematical theory behind any truly efficient factoring algorithm must be both general and unconditional. In addition, it should also be symmetrical. For example, 3D integer factorization searches for a multiple of any prime factor. Since the modulus is a semiprime there is an infinite number of multiples for both prime factors and such is symmetry for an equal probability exists of finding either prime factor. Second, the body of the basic algorithm is symmetrical because it consists entirely on the addition and subtraction of both repunits and different bases. Third, for any modulus m, the respective search experiment is symmetrical due to the fact that all m^2 machines have the same chance of discovering any prime factor. In this paper, we showed a new perspective on the integer factorization problem both in terms of mathematical theory and computer organization. The accepted notion of factoring algorithm was questioned and, as a result, the concept of search experiment was introduced via 3D integer factorization. Nine different semiprimes ranging in size from four to twenty decimal digits were factored using this method. The corresponding factoring times range from zero to eighth seconds on a single computer. However, in order to detect the minimum factoring time, we must use multiple machines in any search experiment. In conclusion, we hope that this work will inspire you to boldly go into the wild and invent an amazing factoring algorithm.

REFERENCES

[1] G. E. Andrews, *Number Theory*, Dover Publications, New York, 1994.

[2] Ronald Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21:120-126, February 1978.

[3] A. K. Lenstra**,** *Integer Factoring*, Designs, Codes and Cryptography 19 (2000), 101-128.

[4] R. Crandall and C. Pomerance, *Prime Numbers: A Computational Perspective*, Springer-Verlag, New York, 2001.

[5] S. S. Wagstaff, Jr., *The Joy of Factoring*, Student Mathematical Library, volume 68, American Mathematical Society, Providence, 2013.

[6] V. C. Hamacher, Z. G. Vranesic and S. G. Zaky, *Computer Organization*, McGraw-Hill Companies, New York, 1996.

[7] L. Rempe-Gillen and R. Waldecker, *Primality Testing for Beginners*, Student Mathematical Library, volume 70, American Mathematical Society, Providence, 2014

[8] J. C. L. da Silva, 3D cryptography, Available from https://www.rainbowofprimes.com