Factoring Large Numbers With Quadratic Sieve
The foundation of the most popular public-key cryptography algorithm in use today, RSA, rests on the difficulty of factoring large integers.
The foundation of the most popular public-key cryptography algorithm in use today, RSA, rests on the difficulty of factoring large integers.

When keys are generated, efficient algorithms are used to generate two very large prime numbers and multiply them together. The person who generated the keys knows these two numbers, but everyone else only knows the product.
The Same Problem ...

The foundation of the most popular public-key cryptography algorithm in use today, RSA, rests on the difficulty of factoring large integers.

When keys are generated, efficient algorithms are used to generate two very large prime numbers and multiply them together. The person who generated the keys knows these two numbers, but everyone else only knows the product.

The product contains enough information to encrypt a message to the person; the two primes allow the recipient to decrypt it. There is no known way to decrypt it without using the primes, but by factoring, we can extract the two prime factors from the product and break the encryption.
At the time that RSA was invented in 1977, factoring integers with as few as 80 decimal digits was intractable; all known algorithms were either too slow or required the number to have a special form. This made even small, 256-bit keys relatively secure.
At the time that RSA was invented in 1977, factoring integers with as few as 80 decimal digits was intractable; all known algorithms were either too slow or required the number to have a special form. This made even small, 256-bit keys relatively secure.

The first major breakthrough was quadratic sieve, a relatively simple factoring algorithm invented by Carl Pomerance in 1981, which can factor numbers in excess of 100 digits. It’s still the best known method for numbers under 110 digits or so.
At the time that RSA was invented in 1977, factoring integers with as few as 80 decimal digits was intractable; all known algorithms were either too slow or required the number to have a special form. This made even small, 256-bit keys relatively secure.

The first major breakthrough was quadratic sieve, a relatively simple factoring algorithm invented by Carl Pomerance in 1981, which can factor numbers in excess of 100 digits. It’s still the best known method for numbers under 110 digits or so.

For larger numbers, the general number field sieve (GNFS) is now used. However, the general number field sieve is extremely complicated, and requires extensive explanation and background for even the most basic implementation. However, GNFS is based on the same fundamental ideas as quadratic sieve, so if factoring the largest numbers in the world is your goal, this is the place to start.
At the time that RSA was invented in 1977, factoring integers with as few as 80 decimal digits was intractable; all known algorithms were either too slow or required the number to have a special form. This made even small, 256-bit keys relatively secure.

The first major breakthrough was quadratic sieve, a relatively simple factoring algorithm invented by Carl Pomerance in 1981, which can factor numbers in excess of 100 digits. It’s still the best known method for numbers under 110 digits or so.

For larger numbers, the general number field sieve (GNFS) is now used. However, the general number field sieve is extremely complicated, and requires extensive explanation and background for even the most basic implementation. However, GNFS is based on the same fundamental ideas as quadratic sieve, so if factoring the largest numbers in the world is your goal, this is the place to start.

We’ll begin by addressing a few problems that at first glance have nothing to do with factoring, then assemble them into a working
Suppose I give you a set of integers and I ask you to find a subset of those integers whose product is a square, if one exists. For example, given the set \{10, 24, 35, 52, 54, 78\}, the product \(24 \times 52 \times 78 = 97344 = 312^2\). The brute-force solution, trying every subset, is too expensive because there are an exponential number of subsets.
Finding a Subset of Integers Whose Product is a Square

Suppose I give you a set of integers and I ask you to find a subset of those integers whose product is a square, if one exists. For example, given the set \{10, 24, 35, 52, 54, 78\}, the product \(24 \times 52 \times 78 = 97344 = 312^2\). The brute-force solution, trying every subset, is too expensive because there are an exponential number of subsets.

We’ll take a different approach based on prime factorizations and linear algebra. First, we factor each of the input numbers into prime factors; for now we will assume that these numbers are easy to factor. For the above example set, we get:

\[
\begin{align*}
10 &= 2 \times 5 \\
24 &= 2^3 \times 3 \\
35 &= 5 \times 7 \\
52 &= 2^2 \times 13 \\
54 &= 2 \times 3^3 \\
78 &= 2 \times 3 \times 13
\end{align*}
\]
Finding a Subset of Integers Whose Product is a Square

When you multiply two numbers written as prime factorizations, you simply add the exponents of the primes used. A number is a square if and only if all the exponents in its prime factorization are even.
When you multiply two numbers written as prime factorizations, you simply add the exponents of the primes used. A number is a square if and only if all the exponents in its prime factorization are even.

Suppose we write the above factorizations as vectors, where the $k^{th}$ entry corresponds to the exponent of the $k$th prime number. We get:

$$
\begin{bmatrix}
1 & 0 & 1 & 0 & 0 & 0 \\
3 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 \\
1 & 3 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1
\end{bmatrix}
$$
Finding a Subset of Integers Whose Product is a Square

When you multiply two numbers written as prime factorizations, you simply add the exponents of the primes used. A number is a square if and only if all the exponents in its prime factorization are even.

Suppose we write the above factorizations as vectors, where the \( k^{th} \) entry corresponds to the exponent of the \( k \)th prime number. We get:

\[
\begin{bmatrix}
1 & 0 & 1 & 0 & 0 & 0 \\
3 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 \\
1 & 3 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1
\end{bmatrix}
\]

Now, multiplying numbers is as simple as adding vectors. If we add rows 2, 4, and 6, we get \([6 \ 2 \ 0 \ 0 \ 0 \ 2]\), which has all even exponents and so must be a square.
Finding a Subset of Integers Whose Product is a Square

In more familiar terms, we want the last bit of each entry in the sum to be zero. But in this case, we don’t need to store all of the numbers above, only the last bit of each number. This gives us the following:

\[
\begin{bmatrix}
1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1
\end{bmatrix}
\]
Moreover, since we’re only interested in last bits, we can perform all our addition using one-bit integers with wraparound semantics (in other words, modulo 2).
Moreover, since we’re only interested in last bits, we can perform all our addition using one-bit integers with wraparound semantics (in other words, modulo 2).

If we add rows 2, 4, and 6 in this way, we get

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]

which is the zero vector. In fact, all squares correspond to the zero vector.
Turning This to a Matrix Problem

If we transpose the above matrix, so that rows become columns, we get this:

\[
\begin{bmatrix}
1 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1
\end{bmatrix}
\]

Call this matrix $A$. If we multiply $A$ by the vector $[0 \ 1 \ 0 \ 1 \ 0 \ 1]$ using one-bit integer arithmetic, we get the zero vector. This tells us precisely which numbers we need to multiply to get a square. So, our goal is to find a nonzero vector $x$ such that $Ax = 0$ (remember, all arithmetic here is with one-bit integers).
Turning This to a Matrix Problem

If we transpose the above matrix, so that rows become columns, we get this:

\[
\begin{bmatrix}
1 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 \\
\end{bmatrix}
\]

Call this matrix \( A \). If we multiply \( A \) by the vector

\[
[0 \ 1 \ 0 \ 1 \ 0 \ 1]
\]

using one-bit integer arithmetic, we get the zero vector. This tells us precisely which numbers we need to multiply to get a square. So, our goal is to find a nonzero vector \( x \) such that \( Ax = 0 \) (remember, all arithmetic here is with one-bit integers).
This is the problem of finding the null space of a matrix, the set of vectors such that $Ax = 0$. The problem can be solved using row reduction. We row reduce the matrix, and then assign values to the free variables in a way that gives us a nonzero solution. The other variables will be determined by these values and the matrix. You probably studied this problem using rational numbers, not one-bit integers, but it turns out row reduction works just as well for these. For example, if we add row 1 to row 3 in the above matrix, we get the following:

$$
\begin{bmatrix}
1 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1
\end{bmatrix}
$$
Continuing with the Linear Algebra

Completing the row reduction, we eventually end up with this matrix:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]
Continuing with the Linear Algebra

Completing the row reduction, we eventually end up with this matrix:

$$
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

If we turn this back into a system of equations and rearrange, we get this:

$$
x_1 = 0
$$

$$
x_2 = -x_5 - x_6
$$

$$
x_3 = 0
$$

$$
x_4 = -x_6
$$
Suppose we choose \( x_5 = 0, \ x_6 = 1 \). From the above equations, it follows that the first four vectors have the values 0, 1, 0, and 1 (remember, one-bit integer arithmetic). This gives us our final vector,

\[
\begin{bmatrix}
0 & 1 & 0 & 1 & 0 & 1
\end{bmatrix}
\]
Suppose we choose $x_5 = 0, x_6 = 1$. From the above equations, it follows that the first four vectors have the values 0, 1, 0, and 1 (remember, one-bit integer arithmetic). This gives us our final vector,

$$[0 \ 1 \ 0 \ 1 \ 0 \ 1]$$

If we were to choose $x_5 = 1$ and $x_6 = 0$ instead, we’d get a different solution:

$$[0 \ 1 \ 0 \ 0 \ 1 \ 0]$$

corresponding to $24 \times 54 = 1296 = 36^2$. 
Moreover, a theorem of linear algebra tells us precisely how many input numbers we need to guarantee that a square can be found: as long as we have more columns than rows, the null space is guaranteed to be nontrivial, so that we have a nonzero solution. In other words, we just need more numbers than prime factors used by those numbers. As this case shows, though, this isn’t a necessary condition.
Continuing with the Linear Algebra

Moreover, a theorem of linear algebra tells us precisely how many input numbers we need to guarantee that a square can be found: as long as we have more columns than rows, the null space is guaranteed to be nontrivial, so that we have a nonzero solution. In other words, we just need more numbers than prime factors used by those numbers. As this case shows, though, this isn’t a necessary condition.

The one remaining problem with this method is that if one of the numbers in our set happens to have very large factors, our matrix will have a large number of rows, which requires a lot of storage and makes row reduction inefficient. To avoid this, we require that the input numbers are $B$-smooth, meaning that they only have small factors less than some integer $B$. This also makes them easy to factor.
This method involves factoring using a difference of squares. You might be wondering what squares have to do with factoring. The connection is the very simple factorization method known as Fermat’s method. Although not efficient in general, it embodies the same basic idea as quadratic sieve and works great for numbers with factors close to their square root.
This method involves factoring using a difference of squares. You might be wondering what squares have to do with factoring. The connection is the very simple factorization method known as Fermat’s method. Although not efficient in general, it embodies the same basic idea as quadratic sieve and works great for numbers with factors close to their square root.

The idea is to find two numbers $a$ and $b$ such that $a^2 - b^2 = n$, the number we wish to factor. If we can do this, simple algebra tells us that $(a + b)(a - b) = n$. If we’re lucky, this is a nontrivial factorization of $n$; if we’re not so lucky, one of them is 1 and the other is $n$. 
Fermat’s Method

The concept behind Fermat’s algorithm is to search for an integer $a$ such that $a^2 - n$ is a square. If we find such an $a$, it follows that:

$$a^2 - (a^2 - n) = n$$
Fermat’s Method

The concept behind Fermat’s algorithm is to search for an integer $a$ such that $a^2 - n$ is a square. If we find such an $a$, it follows that:

$$a^2 - (a^2 - n) = n$$

We have a difference of squares equal to $n$. The search is a straightforward linear search: we begin with the ceiling of the square root of $n$, the smallest possible number such that $a^2 - n$ is positive, and increment $a$ until $a^2 - n$ becomes a square. If this ever happens, we try to factor $n$ as

$$(a - \sqrt{a^2 - n})(a + \sqrt{a^2 - n})$$

If the factorization is trivial, we continue incrementing $a$. 
Example Using Fermat’s Method

Let $n = 5959$; $a$ starts out at 78.
Example Using Fermat’s Method

Let $n = 5959$; $a$ starts out at 78.

The numbers $78^2 - 5959$ and $79^2 - 5959$ are not squares, but $80^2 - 5959 = 441 = 21^2$. 
Example Using Fermat’s Method

Let \( n = 5959 \); \( a \) starts out at 78.

The numbers \( 78^2 - 5959 \) and \( 79^2 - 5959 \) are not squares, but \( 80^2 - 5959 = 441 = 21^2 \).

Hence \( (80 - 21)(80 + 21) = 5959 \), and this gives the nontrivial factorization \( 59 \times 101 = 5959 \).
The reason Fermat’s method is slow is because simply performing a linear search of all possible $a$ hoping that we’ll hit one with $a^2 - n$ square is a poor strategy - there just aren’t that many squares around to hit.
The reason Fermat’s method is slow is because simply performing a linear search of all possible $a$ hoping that we’ll hit one with $a^2 - n$ square is a poor strategy - there just aren’t that many squares around to hit.

A better way of going about it is to proactively compute an $a$ having this property (actually a similar property).
The reason Fermat’s method is slow is because simply performing a linear search of all possible $a$ hoping that we’ll hit one with $a^2 - n$ square is a poor strategy - there just aren’t that many squares around to hit.

A better way of going about it is to proactively compute an $a$ having this property (actually a similar property).

The key is to notice that if we take a number of $a^2 - n$ values, none of which are squares themselves, and multiply them, we may get a square, say $S$. 
The reason Fermat’s method is slow is because simply performing a linear search of all possible $a$ hoping that we’ll hit one with $a^2 - n$ square is a poor strategy - there just aren’t that many squares around to hit.

A better way of going about it is to proactively compute an $a$ having this property (actually a similar property).

The key is to notice that if we take a number of $a^2 - n$ values, none of which are squares themselves, and multiply them, we may get a square, say $S$.

Let $A$ be the product of the corresponding values of $a$. Basic algebra shows that $A^2 - S$ is a multiple of $n$. Hence, $(A - \sqrt{S})(A + \sqrt{S})$ is a factorization of some multiple of $n$; in other words, at least one of these shares a factor with $n$. By computing the greatest common divisor of each with $n$ using Euclid’s algorithm, we can identify this factor. Again, it may be trivial (just $n$ itself); if so we try again with a different square.
Fermat’s Method

All that remains is, given a list of $a^2 - n$ values, to find a subset whose product is a square. But this is precisely an instance of the problem discussed in the last section.
Fermat’s Method

All that remains is, given a list of $a^2 - n$ values, to find a subset whose product is a square. But this is precisely an instance of the problem discussed in the last section.

Unfortunately, recall that the method we came up with there is not efficient for numbers with large factors; the matrix becomes too large. What do we do?
All that remains is, given a list of $a^2 - n$ values, to find a subset whose product is a square. But this is precisely an instance of the problem discussed in the last section.

Unfortunately, recall that the method we came up with there is not efficient for numbers with large factors; the matrix becomes too large. What do we do?

We simply throw away numbers with large factors. Theoretical results show that there are a fairly large number of values in the sequence $a^2 - n$ that are smooth. This gives us a new factoring method that works pretty well up to a point.
An Example

For example, consider the number 90283. If we start a at 301 and increment it up to 360 while computing $a^2 - n$, we get the following values:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>318</td>
<td>921</td>
<td>1526</td>
<td>2133</td>
<td>2742</td>
<td>3353</td>
<td></td>
</tr>
<tr>
<td>3966</td>
<td>4581</td>
<td>5198</td>
<td>5817</td>
<td>6438</td>
<td>7061</td>
<td></td>
</tr>
<tr>
<td>7686</td>
<td>8313</td>
<td>8942</td>
<td>9573</td>
<td>10206</td>
<td>10841</td>
<td></td>
</tr>
<tr>
<td>11478</td>
<td>12117</td>
<td>12758</td>
<td>13401</td>
<td>14046</td>
<td>14693</td>
<td></td>
</tr>
<tr>
<td>15342</td>
<td>15993</td>
<td>16646</td>
<td>17301</td>
<td>17958</td>
<td>18617</td>
<td></td>
</tr>
<tr>
<td>19278</td>
<td>19941</td>
<td>20606</td>
<td>21273</td>
<td>21942</td>
<td>22613</td>
<td></td>
</tr>
<tr>
<td>23286</td>
<td>23961</td>
<td>24638</td>
<td>25317</td>
<td>25998</td>
<td>26681</td>
<td></td>
</tr>
<tr>
<td>27366</td>
<td>28053</td>
<td>28742</td>
<td>29433</td>
<td>30126</td>
<td>30821</td>
<td></td>
</tr>
<tr>
<td>31518</td>
<td>32217</td>
<td>32918</td>
<td>33621</td>
<td>34326</td>
<td>35033</td>
<td></td>
</tr>
<tr>
<td>35742</td>
<td>36453</td>
<td>37166</td>
<td>37881</td>
<td>38598</td>
<td>39317</td>
<td></td>
</tr>
</tbody>
</table>
None of these are squares (the first square occurs at $a = 398$); however, if we factor each value we will discover that 7 of these values have no factor larger than 43:

$$6438, 10206, 16646, 19278, 19941, 30821, 35742$$
None of these are squares (the first square occurs at $a = 398$); however, if we factor each value we will discover that 7 of these values have no factor larger than 43:

$$6438, 10206, 16646, 19278, 19941, 30821, 35742$$

If we take these 7 values and feed them to the algorithm described earlier, it finds a square:

$$19278 \times 19941 \times 30821 \times 35742 = 423481541612104836 = 650754594^2$$

The corresponding original $a$ were 331, 332, 348, and 355, and their product is 13576057680.
An Example

Now, we can factor the number:

$$(13576057680 - 650754594)(13576057680 + 650754594)$$

$$= 12925303086 \times 14226812274$$

is a multiple of 90283.
Now, we can factor the number:

\[(13576057680 - 650754594)(13576057680 + 650754594)\]
\[= 12925303086 \times 14226812274\]

is a multiple of 90283.

\[(90283, 12925303086) = 137\]
\[(90283, 14226812274) = 659\]

\[137 \times 659 = 90283\]
The factorization algorithm above is considerably better than Fermat’s algorithm, but if we try to scale up the size of number we factor, we quickly encounter a bottleneck: finding the smooth numbers in the sequence.
The factorization algorithm above is considerably better than Fermat’s algorithm, but if we try to scale up the size of number we factor, we quickly encounter a bottleneck: finding the smooth numbers in the sequence.

Only 7 of the 60 values we computed in our last example were 43-smooth (actually we were lucky to get a square with so few vectors). As the size of the number that we’re factoring grows, so does the size of the numbers in the sequence, and the proportion of smooth numbers rapidly shrinks.
Sieving for Smooth Numbers

The factorization algorithm above is considerably better than Fermat’s algorithm, but if we try to scale up the size of number we factor, we quickly encounter a bottleneck: finding the smooth numbers in the sequence.

Only 7 of the 60 values we computed in our last example were 43-smooth (actually we were lucky to get a square with so few vectors). As the size of the number that we’re factoring grows, so does the size of the numbers in the sequence, and the proportion of smooth numbers rapidly shrinks.

Although finding smooth numbers doesn’t require completely factoring every number in the sequence (we only have to test primes up to the smoothness limit), it’s still too expensive to test every number in the sequence this way.
Being Aware of Prime Factors

The key is to observe that the prime factors of the sequence $a^2 - n$ follow a predictable sequence. Let’s take a look at the prime factorizations of the first ten or so numbers in our example sequence above:

\[
\begin{align*}
318 &= 2 \times 3 \times 53 \\
921 &= 3 \times 307 \\
1526 &= 2 \times 7 \times 109 \\
2133 &= 3^3 \times 79 \\
2742 &= 2 \times 3 \times 457 \\
3353 &= 7 \times 479 \\
3966 &= 2 \times 3 \times 661 \\
4581 &= 3^2 \times 509 \\
5198 &= 2 \times 23 \times 113 \\
5817 &= 3 \times 7 \times 277
\end{align*}
\]
Observations

The most obvious pattern is that every other number is even, beginning with the first one. This should be no surprise, since we’re effectively adding $2a + 1$ to get each new number, which is always odd.
The most obvious pattern is that every other number is even, beginning with the first one. This should be no surprise, since we’re effectively adding $2a + 1$ to get each new number, which is always odd.

Also, you’ll notice that the first and second numbers are divisible by 3, as are the fourth and fifth, the seventh and eighth, and so on.
Observations

The most obvious pattern is that every other number is even, beginning with the first one. This should be no surprise, since we’re effectively adding $2a + 1$ to get each new number, which is always odd.

Also, you’ll notice that the first and second numbers are divisible by 3, as are the fourth and fifth, the seventh and eighth, and so on.

If you look at the larger list, you’ll notice similar patterns for larger primes; for example, the 3rd and 6th numbers are divisible by 7, and every 7th number after each of them as well. And, mysteriously, not one number in our entire sequence is divisible by 5 ...
So what’s going on? The answer involves what number theorists call quadratic residues.

### Definition
A number $a$ is called a **quadratic residue** (modulo $p$) if there is some square $S$ such that $S - a$ is divisible by $p$.

Half of all numbers are quadratic residues modulo $p$, regardless of the value of $p$, and there's a simple formula for determining whether or not a particular number is: just take $a$, raise it to the power $p - 1/2$, and then take the remainder after division by $p$. Then $a$ is a quadratic residue modulo $p$ if and only if the answer is 1.

Although this computation seems to involve very large values, in fact we can compute it quite quickly using exponentiation by squaring combined with frequent remainder operations.
So what’s going on? The answer involves what number theorists call quadratic residues.

**Definition**

A number $a$ is called a **quadratic residue** (modulo $p$) if there is some square $S$ such that $S - a$ is divisible by $p$. 

Half of all numbers are quadratic residues modulo $p$, regardless of the value of $p$, and there’s a simple formula for determining whether or not a particular number is: just take $a$, raise it to the power $p - 1/2$, and then take the remainder after division by $p$. Then $a$ is a quadratic residue modulo $p$ if and only if the answer is 1. Although this computation seems to involve very large values, in fact we can compute it quite quickly using exponentiation by squaring combined with frequent remainder operations.
Quadratic Residues

So what’s going on? The answer involves what number theorists call quadratic residues.

**Definition**

A number $a$ is called a **quadratic residue** (modulo $p$) if there is some square $S$ such that $S - a$ is divisible by $p$.

Half of all numbers are quadratic residues modulo $p$, regardless of the value of $p$, and there’s a simple formula for determining whether or not a particular number is: just take $a$, raise it to the power $\frac{p-1}{2}$, and then take the remainder after division by $p$. Then $a$ is a quadratic residue modulo $p$ if and only if the answer is 1.
Quadratic Residues

So what’s going on? The answer involves what number theorists call quadratic residues.

**Definition**

A number $a$ is called a \textbf{quadratic residue} (modulo $p$) if there is some square $S$ such that $S - a$ is divisible by $p$.

Half of all numbers are quadratic residues modulo $p$, regardless of the value of $p$, and there’s a simple formula for determining whether or not a particular number is: just take $a$, raise it to the power $\frac{p-1}{2}$, and then take the remainder after division by $p$. Then $a$ is a quadratic residue modulo $p$ if and only if the answer is 1.

Although this computation seems to involve very large values, in fact we can compute it quite quickly using exponentiation by squaring combined with frequent remainder operations.
If we compute $90283^{\frac{5-1}{2}} \pmod{5}$, we get 4, which is not 1 (remember that 90283 is our original $n$ to be factored). Thus, there is no square such that $S - n$ is divisible by 5.
If we compute $90283 \frac{5-1}{2} \pmod{5}$, we get 4, which is not 1 (remember that 90283 is our original $n$ to be factored). Thus, there is no square such that $S - n$ is divisible by 5.

All numbers in our sequence have this form. In practice, this means we can compute just once ahead of time which factors may occur in the sequence (primes $p$ such that $n$ is a quadratic residue modulo $p$), and ignore all others.
Every $p^{th}$ Number is Divisible By $p$?

For our next mystery, why is it that given a number in the sequence divisible by $p$, every $p^{th}$ number after that is also divisible by $p$? Well, simple algebra shows that if $a^2 - n = kp$, then:

$$(a + p)^2 - n = (a^2 - n) + p(2a + p) = kp + p(2a + p).$$
Every $p^{th}$ Number is Divisible By $p$?

For our next mystery, why is it that given a number in the sequence divisible by $p$, every $p^{th}$ number after that is also divisible by $p$? Well, simple algebra shows that if $a^2 - n = kp$, then:

$$(a + p)^2 - n = (a^2 - n) + p(2a + p) = kp + p(2a + p).$$

But this doesn’t explain why it always seems to be the case that there are exactly two different initial values of $a$ such that $a^2 - n$ is divisible by $p$ (with the exception of $p = 2$).
Every $p^{th}$ Number is Divisible By $p$?

For our next mystery, why is it that given a number in the sequence divisible by $p$, every $p^{th}$ number after that is also divisible by $p$? Well, simple algebra shows that if $a^2 - n = kp$, then:

$$(a + p)^2 - n = (a^2 - n) + p(2a + p) = kp + p(2a + p).$$

But this doesn’t explain why it always seems to be the case that there are exactly two different initial values of $a$ such that $a^2 - n$ is divisible by $p$ (with the exception of $p = 2$).

For example, in our sequence above the 3$^{rd}$ and 6$^{th}$ values were divisible by 7. The answer again is quadratic residues: it can be shown that the modular equation $x^2 \equiv y \pmod{p}$ has exactly two solutions (if it has any), and in fact there is an efficient algorithm for computing these two solutions called the Tonelli-Shanks algorithm.
The Tonelli-Shanks algorithm is used within modular arithmetic to solve a congruence of the form

\[ x^2 \equiv n \pmod{p} \]

where \( n \) is a quadratic residue modulo \( p \) and \( p \) is an odd prime.
The Tonelli-Shanks algorithm is used within modular arithmetic to solve a congruence of the form

$$x^2 \equiv n \pmod{p}$$

where $n$ is a quadratic residue modulo $p$ and $p$ is an odd prime.

Note: Tonelli-Shanks cannot be used for composite moduli; finding square roots modulo composite numbers is a computational problem equivalent to integer factorization.
Tonelli-Shanks Algorithm

- **Input**
  - $p$, an odd prime
  - $n$, an integer which is a quadratic residue modulo $p$, meaning that the Legendre symbol $\left( \frac{n}{p} \right) = 1$.
Tonelli-Shanks Algorithm

- **Input**
  - \(p\), an odd prime
  - \(n\), an integer which is a quadratic residue modulo \(p\), meaning that the Legendre symbol \(\left( \frac{n}{p} \right) = 1\).

**Definition**

Let \(p\) be an odd prime number. An integer \(a\) is a quadratic residue modulo \(p\) if it is congruent to a perfect square modulo \(p\) and a quadratic nonresidue otherwise. The **Legendre symbol** is a function of \(a\) and \(p\) defined as follows:

\[
\left( \frac{a}{p} \right) = \begin{cases} 
  1 & \text{if } a \text{ is a quadratic residue modulo } p \text{ and } a \not\equiv 0 \pmod{p} \\
  -1 & \text{if } a \text{ is a quadratic nonresidue modulo } p \\
  0 & \text{if } a \equiv 0 \pmod{p}
\end{cases}
\]
Tonelli-Shanks Algorithm

- Input
  - $p$, an odd prime
  - $n$, an integer which is a quadratic residue modulo $p$, meaning that the Legendre symbol $\left( \frac{n}{p} \right) = 1$.

Definition

Let $p$ be an odd prime number. An integer $a$ is a quadratic residue modulo $p$ if it is congruent to a perfect square modulo $p$ and a quadratic nonresidue otherwise. The **Legendre symbol** is a function of $a$ and $p$ defined as follows:

$$\left( \frac{a}{p} \right) = \begin{cases} 1 & \text{if } a \text{ is a quadratic residue modulo } p \text{ and } a \not\equiv 0 \pmod{p} \\ -1 & \text{if } a \text{ is a quadratic nonresidue modulo } p \\ 0 & \text{if } a \equiv 0 \pmod{p} \end{cases}$$

- Output
  - $R$, an integer satisfying $R^2 \equiv n \pmod{p}$. 
Factor out the powers of 2 from $p - 1$, defining $Q$ and $S$ as

$$p - 1 = Q \cdot 2^S$$

with $Q$ odd. Note that if $S = 1$, i.e. $p \equiv 3 \pmod{4}$, then the solutions are given directly by

$$R \equiv \pm n^{\frac{p+1}{4}}$$

Select a $z$ such that the Legendre symbol $\left(\frac{z}{p}\right) = -1$ (that is, $z$ should be a quadratic nonresidue modulo $p$) and set $c \equiv z^Q$. 

Loop:

1. If $t \equiv 1$, return $R$.
2. Otherwise, find the lowest $i$, $0 < i < M$ such that $t^{2^i} \equiv 1$

Let $b \equiv c^{2M-i-1}$ and set $R = Rb$, $t \equiv tb^2$, $c = b$ and $M = i$. Once you have solved the congruence with $R$, the second solution is $p - R$. 

Tonelli-Shanks Algorithm
Factor out the powers of 2 from \( p - 1 \), defining \( Q \) and \( S \) as

\[
p - 1 = Q \cdot 2^S
\]

with \( Q \) odd. Note that if \( S = 1 \), i.e. \( p \equiv 3 \pmod{4} \), then the solutions are given directly by

\[
R \equiv \pm n^{\frac{p+1}{4}}
\]

Select a \( z \) such that the Legendre symbol \( \left( \frac{z}{p} \right) = -1 \) (that is, \( z \) should be a quadratic nonresidue modulo \( p \)) and set \( c \equiv z^Q \).

Let \( R \equiv n^{\frac{Q+1}{2}} \), \( t \equiv n^Q \), \( M = S \).
Tonelli-Shanks Algorithm

1. Factor out the powers of 2 from $p - 1$, defining $Q$ and $S$ as

$$p - 1 = Q \cdot 2^S$$

with $Q$ odd. Note that if $S = 1$, i.e. $p \equiv 3 \pmod{4}$, then the solutions are given directly by

$$R \equiv \pm n^{\frac{p+1}{4}}$$

Select a $z$ such that the Legendre symbol $\left( \frac{z}{p} \right) = -1$ (that is, $z$ should be a quadratic nonresidue modulo $p$) and set $c \equiv z^Q$.

2. Let $R \equiv n^{\frac{Q+1}{2}}$, $t \equiv n^Q$, $M = S$.

3. Loop:

   1. If $t \equiv 1$, return $R$. 
Factor out the powers of 2 from $p - 1$, defining $Q$ and $S$ as

$$p - 1 = Q \cdot 2^S$$

with $Q$ odd. Note that if $S = 1$, i.e. $p \equiv 3 \pmod{4}$, then the solutions are given directly by

$$R \equiv \pm n^{\frac{p+1}{4}}$$

Select a $z$ such that the Legendre symbol $\left( \frac{z}{p} \right) = -1$ (that is, $z$ should be a quadratic nonresidue modulo $p$) and set $c \equiv z^Q$.

Let $R \equiv n^{\frac{Q+1}{2}}$, $t \equiv n^Q$, $M = S$.

Loop:

1. If $t \equiv 1$, return $R$.
2. Otherwise, find the lowest $i$, $0 < i < M$ such that $t^{2i} \equiv 1$
Factor out the powers of 2 from \( p - 1 \), defining \( Q \) and \( S \) as
\[
p - 1 = Q \cdot 2^S
\]
with \( Q \) odd. Note that if \( S = 1 \), i.e. \( p \equiv 3 \pmod{4} \), then the solutions are given directly by
\[
R \equiv \pm n^{\frac{p+1}{4}}
\]
Select a \( z \) such that the Legendre symbol \( \left( \frac{z}{p} \right) = -1 \) (that is, \( z \) should be a quadratic nonresidue modulo \( p \)) and set \( c \equiv z^Q \).

Let \( R \equiv n^{\frac{Q+1}{2}} \), \( t \equiv n^Q \), \( M \equiv S \).

Loop:
1. If \( t \equiv 1 \), return \( R \).
2. Otherwise, find the lowest \( i \), \( 0 < i < M \) such that \( t^{2i} \equiv 1 \)
3. Let \( b \equiv c^{2^{M-i-1}} \) and set \( R = Rb \), \( t \equiv tb^2 \), \( c = b^2 \) and \( M = i \).
1. Factor out the powers of 2 from \( p - 1 \), defining \( Q \) and \( S \) as

\[
p - 1 = Q \cdot 2^S
\]

with \( Q \) odd. Note that if \( S = 1 \), i.e. \( p \equiv 3 \pmod{4} \), then the solutions are given directly by

\[
R \equiv \pm n^{\frac{p+1}{4}}
\]

Select a \( z \) such that the Legendre symbol \( \left( \frac{z}{p} \right) = -1 \) (that is, \( z \) should be a quadratic nonresidue modulo \( p \)) and set \( c \equiv z^Q \).

2. Let \( R \equiv n^{\frac{Q+1}{2}} \), \( t \equiv n^Q \), \( M = S \).

3. Loop:
   1. If \( t \equiv 1 \), return \( R \).
   2. Otherwise, find the lowest \( i \), \( 0 < i < M \) such that \( t^{2i} \equiv 1 \)
   3. Let \( b \equiv c^{2^{M-i-1}} \) and set \( R = Rb \), \( t \equiv tb^2 \), \( c = b^2 \) and \( M = i \).

Once you have solved the congruence with \( R \), the second solution is \( p - R \).
Example Using Tonelli-Shanks

Example

Solve the congruence $x^2 \equiv 10 \pmod{13}$ using Tonelli-Shanks.

It is clear that 13 is odd, and since

$$10^{\frac{13-1}{2}} = 10^6 \equiv 1 \pmod{13}$$

10 is a quadratic residue.

Step 1: Observe $p - 1 = 12 = 3 \cdot 2^2$, so $Q = 3$ and $S = 2$. 
Example Using Tonelli-Shanks

Example

Solve the congruence $x^2 \equiv 10 \pmod{13}$ using Tonelli-Shanks.

It is clear that 13 is odd, and since

$$10^{\frac{13-1}{2}} = 10^6 \equiv 1 \pmod{13}$$

10 is a quadratic residue.

Step 1: Observe $p - 1 = 12 = 3 \cdot 2^2$, so $Q = 3$ and $S = 2$.

Step 2: Take $z = 2$ as the quadratic nonresidue by Euler’s criterion since

$$2^{\frac{13-1}{2}} \equiv -1 \pmod{13}$$

Set $c = 2^3 \equiv 8 \pmod{13}$. 
Example Using Tonelli-Shanks

Example

Solve the congruence $x^2 \equiv 10 \pmod{13}$ using Tonelli-Shanks.

It is clear that 13 is odd, and since

$$10^{\frac{13-1}{2}} = 10^6 \equiv 1 \pmod{13}$$

10 is a quadratic residue.

Step 1: Observe $p - 1 = 12 = 3 \cdot 2^2$, so $Q = 3$ and $S = 2$.

Step 2: Take $z = 2$ as the quadratic nonresidue by Euler’s criterion since

$$2^{\frac{13-1}{2}} \equiv -1 \pmod{13}$$

Set $c = 2^3 \equiv 8 \pmod{13}$.

Step 3: $R = 10^2 \equiv -4$, $t = 10^3 \equiv -1 \pmod{13}$, $M = 2$
Step 4: Now we start the loop:

- \( t \not\equiv 1 \pmod{13} \), so \( 0 < i < 2 \); i.e., \( i = 1 \)
Example Using Tonelli-Shanks

Step 4: Now we start the loop:

- $t \not\equiv 1 \pmod{13}$, so $0 < i < 2$; i.e., $i = 1$
- Let $b \equiv 8^{2^{i-1}} \equiv 8 \pmod{13}$, so $b^2 \equiv 8^2 \equiv -1 \pmod{13}$
Step 4: Now we start the loop:

- $t \not\equiv 1 \pmod{13}$, so $0 < i < 2$; i.e., $i = 1$
- Let $b \equiv 8^{2^2-1-1} \equiv 8 \pmod{13}$, so $b^2 \equiv 8^2 \equiv -1 \pmod{13}$
- Set $R = -4 \cdot 8 \equiv 7 \pmod{13}$. Set $t \equiv -1 \cdot -1 \equiv 1 \pmod{13}$ and $M = 1$. We restart the loop and since $t \equiv 1 \pmod{13}$. We are done, returning $R \equiv 7 \pmod{13}$
Step 4: Now we start the loop:

- $t \not\equiv 1 \pmod{13}$, so $0 < i < 2$; i.e., $i = 1$
- Let $b \equiv 8^{2i-1} \equiv 8 \pmod{13}$, so $b^2 \equiv 8^2 \equiv -1 \pmod{13}$
- Set $R = -4 \cdot 8 \equiv 7 \pmod{13}$. Set $t \equiv -1 \cdot -1 \equiv 1 \pmod{13}$ and $M = 1$. We restart the loop and since $t \equiv 1 \pmod{13}$. We are done, returning $R \equiv 7 \pmod{13}$

Notice that $7^2 = 49 \equiv 10 \pmod{13}$ and $(-7)^2 \equiv 6^2 \equiv 10 \pmod{13}$. So, the algorithm yields two solutions to our congruence.
It suffices to test the first $p$ numbers to see which are divisible by $p$. For larger primes, it becomes important to avoid this expensive scan.
It suffices to test the first $p$ numbers to see which are divisible by $p$. For larger primes, it becomes important to avoid this expensive scan.

Recall the Sieve of Eratosthenes, an algorithm for locating prime numbers. It starts with a list of numbers, then crosses off all numbers not divisible by 2 except 2, then does the same for 3, 5, and so on until it’s done. The numbers that remain must be prime. When attempting to find a list of prime numbers, this strategy is much more efficient than running even the most advanced primality test on each number individually.
We take a similar strategy here: we begin with a table of the original values in the sequence. We then visit all the numbers divisible by 2 and divide out a factor of 2. We do the same for each power of 2 up to the size of the sequence.
Our Strategy

We take a similar strategy here: we begin with a table of the original values in the sequence. We then visit all the numbers divisible by 2 and divide out a factor of 2. We do the same for each power of 2 up to the size of the sequence.

We then do the same for every other prime up to our smoothness bound (43 in our example). In the end, the smooth numbers and only the smooth numbers will have become 1. Since we visit less and less list elements as the prime factor increases, the overall work is much less.
For example, here’s our original list from the above example:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>318</td>
<td>921</td>
<td>1526</td>
<td>2133</td>
<td>2742</td>
<td>3353</td>
<td></td>
</tr>
<tr>
<td>3966</td>
<td>4581</td>
<td>5198</td>
<td>5817</td>
<td>6438</td>
<td>7061</td>
<td></td>
</tr>
<tr>
<td>7686</td>
<td>8313</td>
<td>8942</td>
<td>9573</td>
<td>10206</td>
<td>10841</td>
<td></td>
</tr>
<tr>
<td>11478</td>
<td>12117</td>
<td>12758</td>
<td>13401</td>
<td>14046</td>
<td>14693</td>
<td></td>
</tr>
<tr>
<td>15342</td>
<td>15993</td>
<td>16646</td>
<td>17301</td>
<td>17958</td>
<td>18617</td>
<td></td>
</tr>
<tr>
<td>19278</td>
<td>19941</td>
<td>20606</td>
<td>21273</td>
<td>21942</td>
<td>22613</td>
<td></td>
</tr>
<tr>
<td>23286</td>
<td>23961</td>
<td>24638</td>
<td>25317</td>
<td>25998</td>
<td>26681</td>
<td></td>
</tr>
<tr>
<td>27366</td>
<td>28053</td>
<td>28742</td>
<td>29433</td>
<td>30126</td>
<td>30821</td>
<td></td>
</tr>
<tr>
<td>31518</td>
<td>32217</td>
<td>32918</td>
<td>33621</td>
<td>34326</td>
<td>35033</td>
<td></td>
</tr>
<tr>
<td>35742</td>
<td>36453</td>
<td>37166</td>
<td>37881</td>
<td>38598</td>
<td>39317</td>
<td></td>
</tr>
</tbody>
</table>
Our Example and the Sieve

We visit elements 1, 3, 5, and so on, dividing out 2. Here’s the list after this first pass is complete:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>159</td>
<td>921</td>
<td>763</td>
<td>2133</td>
<td>1371</td>
<td>3353</td>
<td></td>
</tr>
<tr>
<td>1983</td>
<td>4581</td>
<td>2599</td>
<td>5817</td>
<td>3219</td>
<td>7061</td>
<td></td>
</tr>
<tr>
<td>3843</td>
<td>8313</td>
<td>4471</td>
<td>9573</td>
<td>5103</td>
<td>10841</td>
<td></td>
</tr>
<tr>
<td>5739</td>
<td>12117</td>
<td>6379</td>
<td>13401</td>
<td>7023</td>
<td>14693</td>
<td></td>
</tr>
<tr>
<td>7671</td>
<td>15993</td>
<td>8323</td>
<td>17301</td>
<td>8979</td>
<td>18617</td>
<td></td>
</tr>
<tr>
<td>9639</td>
<td>19941</td>
<td>10303</td>
<td>21273</td>
<td>10971</td>
<td>22613</td>
<td></td>
</tr>
<tr>
<td>11643</td>
<td>23961</td>
<td>12319</td>
<td>25317</td>
<td>12999</td>
<td>26681</td>
<td></td>
</tr>
<tr>
<td>13683</td>
<td>28053</td>
<td>14371</td>
<td>29433</td>
<td>15063</td>
<td>30821</td>
<td></td>
</tr>
<tr>
<td>15759</td>
<td>32217</td>
<td>16459</td>
<td>33621</td>
<td>17163</td>
<td>35033</td>
<td></td>
</tr>
<tr>
<td>17871</td>
<td>36453</td>
<td>18583</td>
<td>37881</td>
<td>19299</td>
<td>39317</td>
<td></td>
</tr>
</tbody>
</table>
Here it is after dividing out the prime factors 3, 5, 7, 11, 13, and 17:

<table>
<thead>
<tr>
<th></th>
<th>53</th>
<th>307</th>
<th>109</th>
<th>79</th>
<th>457</th>
<th>479</th>
</tr>
</thead>
<tbody>
<tr>
<td>661</td>
<td>509</td>
<td>2599</td>
<td>277</td>
<td>1073</td>
<td>7061</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>163</td>
<td>263</td>
<td>3191</td>
<td>1</td>
<td>10841</td>
<td></td>
</tr>
<tr>
<td>1913</td>
<td>577</td>
<td>6379</td>
<td>1489</td>
<td>2341</td>
<td>2099</td>
<td></td>
</tr>
<tr>
<td>2557</td>
<td>1777</td>
<td>1189</td>
<td>5767</td>
<td>2993</td>
<td>18617</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>23</td>
<td>10303</td>
<td>1013</td>
<td>1219</td>
<td>22613</td>
<td></td>
</tr>
<tr>
<td>3881</td>
<td>163</td>
<td>12319</td>
<td>2813</td>
<td>619</td>
<td>26681</td>
<td></td>
</tr>
<tr>
<td>4561</td>
<td>1039</td>
<td>2053</td>
<td>9811</td>
<td>5021</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>10739</td>
<td>16459</td>
<td>1601</td>
<td>1907</td>
<td>35033</td>
<td></td>
</tr>
<tr>
<td>851</td>
<td>12151</td>
<td>18583</td>
<td>1403</td>
<td>919</td>
<td>39317</td>
<td></td>
</tr>
</tbody>
</table>
Our Example and the Sieve

We see a couple 1s have already appeared; these are 17-smooth numbers. When we get all the way up through 43, we have:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>307</td>
<td>109</td>
<td>79</td>
<td>457</td>
<td>479</td>
<td>661</td>
</tr>
<tr>
<td>661</td>
<td>509</td>
<td>113</td>
<td>277</td>
<td>1</td>
<td>307</td>
<td>61</td>
</tr>
<tr>
<td>61</td>
<td>163</td>
<td>263</td>
<td>3191</td>
<td>1</td>
<td>293</td>
<td></td>
</tr>
<tr>
<td>1913</td>
<td>577</td>
<td>6379</td>
<td>1489</td>
<td>2341</td>
<td>2099</td>
<td></td>
</tr>
<tr>
<td>2557</td>
<td>1777</td>
<td>1</td>
<td>5767</td>
<td>73</td>
<td>18617</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10303</td>
<td>1013</td>
<td>53</td>
<td>22613</td>
<td></td>
</tr>
<tr>
<td>3881</td>
<td>163</td>
<td>12319</td>
<td>97</td>
<td>619</td>
<td>26681</td>
<td></td>
</tr>
<tr>
<td>4561</td>
<td>1039</td>
<td>2053</td>
<td>9811</td>
<td>5021</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>10739</td>
<td>16459</td>
<td>1601</td>
<td>1907</td>
<td>35033</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>419</td>
<td>18583</td>
<td>61</td>
<td>919</td>
<td>39317</td>
<td></td>
</tr>
</tbody>
</table>

We see several numbers set to 53 or 61; these would be smooth if we raised our bound a little bit.
Sieving Example

This sieving process is where quadratic sieve gets its name from. This drastically decreases the overall work needed to find a sufficient number of smooth numbers, making it practical for very large numbers. This basic implementation could probably handle numbers up to 50-60 digits.
This sieving process is where quadratic sieve gets its name from. This drastically decreases the overall work needed to find a sufficient number of smooth numbers, making it practical for very large numbers. This basic implementation could probably handle numbers up to 50-60 digits.

**Example**

Factor $n = 16843009$ using the quadratic sieve algorithm with a factor base of $B = \{2, 3, 5, 7, 11, 13\}$. 
First, the nearest square root of $n$ is 4104. We are searching for two terms of the form $a^2 - n$ that are $B$-smooth and so that the sum of the powers of these primes is even.

<table>
<thead>
<tr>
<th>$a$</th>
<th>$a^2 - n$</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>4122</td>
<td>147875</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4159</td>
<td>454272</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4187</td>
<td>687960</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4241</td>
<td>1143072</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4497</td>
<td>3380000</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
We see that for the last two values of $a$, the sum of the powers is even so we have

$$a_1 = 1143072 = 2^5 \cdot 3^6 \cdot 5^0 \cdot 7^2 \cdot 11^0 \cdot 13^0$$
$$a_2 = 3380000 = 2^5 \cdot 3^0 \cdot 5^4 \cdot 7^0 \cdot 11^0 \cdot 13^2$$
We see that for the last two values of $a$, the sum of the powers is even so we have

\[ a_1 = 1143072 = 2^5 \cdot 3^6 \cdot 5^0 \cdot 7^2 \cdot 11^0 \cdot 13^0 \]
\[ a_2 = 3380000 = 2^5 \cdot 3^0 \cdot 5^4 \cdot 7^0 \cdot 11^0 \cdot 13^2 \]

This gives

\[ 4241 \cdot 4497 = 19071777 \]
\[ 1143072 \cdot 3380000 = 2^5 \cdot 3^3 \cdot 5^2 \cdot 7^1 \cdot 11^0 \cdot 13^1 = 1965600 \]
Sieving Example

We see that for the last two values of $a$, the sum of the powers is even so we have

$$a_1 = 1143072 = 2^5 \cdot 3^6 \cdot 5^0 \cdot 7^2 \cdot 11^0 \cdot 13^0$$
$$a_2 = 3380000 = 2^5 \cdot 3^0 \cdot 5^4 \cdot 7^0 \cdot 11^0 \cdot 13^2$$

This gives

$$4241 \cdot 4497 = 19071777$$
$$1143072 \cdot 3380000 = 2^5 \cdot 3^3 \cdot 5^2 \cdot 7^1 \cdot 11^0 \cdot 13^1 = 1965600$$

Now, we need to find the greatest common factor of the sum and difference of these two with $n = 16843009$.

$$(19071777 - 1965600, 16843009) = 257$$
$$(19071777 + 1965600, 16843009) = 65537$$
Improvements and Optimizations

Quadratic sieve admits a number of ‘bells and whistles’ to dramatically improve its runtime in practice.
Improvements and Optimizations

Quadratic sieve admits a number of ‘bells and whistles’ to dramatically improve its runtime in practice.

The simple row reduction method of Gaussian elimination is not able to accommodate the very large smoothness bounds needed to factor large numbers, which often range in the millions, mostly due to space limitations; such matrices, if stored explicitly, would require trillions of bits.

However, this method is wasteful, because most of the entries in the matrix are zero (they must be; each number has no more than \( \log_2 n \) prime factors). Instead of using an actual two-dimensional array, we can just keep a list for each column that lists the positions of the 1 bits in that column. We then use a method well-suited to reducing sparse matrices such as the Lanczos algorithm.
Improvements and Optimizations

Quadratic sieve admits a number of ‘bells and whistles’ to dramatically improve its runtime in practice.

The simple row reduction method of Gaussian elimination is not able to accommodate the very large smoothness bounds needed to factor large numbers, which often range in the millions, mostly due to space limitations; such matrices, if stored explicitly, would require trillions of bits.

However, this method is wasteful, because most of the entries in the matrix are zero (they must be; each number has no more than \( \log_2 n \) prime factors).
Quadratic sieve admits a number of ‘bells and whistles’ to dramatically improve its runtime in practice.

The simple row reduction method of Gaussian elimination is not able to accommodate the very large smoothness bounds needed to factor large numbers, which often range in the millions, mostly due to space limitations; such matrices, if stored explicitly, would require trillions of bits.

However, this method is wasteful, because most of the entries in the matrix are zero (they must be; each number has no more than $\log_2 n$ prime factors).

Instead of using an actual two-dimensional array, we can just keep a list for each column that lists the positions of the 1 bits in that column. We then use a method well-suited to reducing sparse matrices such as the Lanczos algorithm.
The Lanczos algorithm is an iterative algorithm devised by Cornelius Lanczos that is an adaptation of power methods to find eigenvalues and eigenvectors of a square matrix or the singular value decomposition of a rectangular matrix. It is particularly useful for finding decompositions of very large sparse matrices. In latent semantic indexing, for instance, matrices relating millions of documents to hundreds of thousands of terms must be reduced to singular-value form.
Improvements and Optimizations

This still requires a fair amount of space; it’s common to use block algorithms that work on small portions of the matrix at one time, storing the rest of the matrix on disk. The matrix step is notoriously difficult to parallelize and for large problems is often done on a single high-performance supercomputer.
This still requires a fair amount of space; it’s common to use block algorithms that work on small portions of the matrix at one time, storing the rest of the matrix on disk. The matrix step is notoriously difficult to parallelize and for large problems is often done on a single high-performance supercomputer.

The most expensive step by far is the sieving, which can require scanning billions of numbers to locate the needed smooth numbers. A common trick is to only track the approximate logarithm of each number, usually in fixed-point arithmetic. Then, when visiting each number, instead of performing an expensive division we only have to subtract.
This introduces a bit of rounding error into the algorithm, but that’s okay; by rounding consistently in the correct direction, we can ensure that we don’t miss any smooth numbers and only capture a few spurious numbers that we can quickly check and reject. Because the logarithms of small primes are small, and require visiting more numbers than any others, primes like 2 and 3 are often dropped altogether.
Another problem is that $a^2 - n$ grows fairly quickly; because smaller numbers are more likely to be smooth, we get diminishing returns as we scan higher in the sequence.
Another problem is that $a^2 - n$ grows fairly quickly; because smaller numbers are more likely to be smooth, we get diminishing returns as we scan higher in the sequence.

To get around this, we scan values of not just the sequence $a^2 - n$ but also a number of similar sequences such as $(Ca + b)^2 - n$ for suitable constants $C, b$. This variation is called the multiple polynomial quadratic sieve, since each of these sequences can be seen as the values of polynomial in $a$. 
Another problem is that $a^2 - n$ grows fairly quickly; because smaller numbers are more likely to be smooth, we get diminishing returns as we scan higher in the sequence.

To get around this, we scan values of not just the sequence $a^2 - n$ but also a number of similar sequences such as $(Ca + b)^2 - n$ for suitable constants $C, b$. This variation is called the multiple polynomial quadratic sieve, since each of these sequences can be seen as the values of polynomial in $a$.

Finally, although the matrix step does not admit simple parallelization due to many data dependencies, the sieving step is perfectly suited to massive parallelization.
So What Is Parallelization?

The problem: Given a matrix $A_{m,r}$, where each of its elements is denoted $a_{ij}$ with $1 \leq i \leq m$ and $1 \leq j \leq r$, and a matrix $B_{r,n}$, where each of its elements is denoted $b_{ij}$ with $1 \leq i \leq r$, and $1 \leq j \leq n$, the matrix $C$ resulting from the operation of multiplication of matrices $A$ and $B$, $C = A \times B$, is such that each of its elements is denoted $c_{ij}$ with $1 \leq i \leq m$ and $1 \leq j \leq n$, and is calculated as follows

$$c_{ij} = \sum_{k=1}^{r} a_{ik}b_{kj}$$
The problem: Given a matrix $A_{m,r}$, where each of its elements is denoted $a_{ij}$ with $1 \leq i \leq m$ and $1 \leq j \leq r$, and a matrix $B_{r,n}$, where each of its elements is denoted $b_{ij}$ with $1 \leq i \leq r$, and $1 \leq j \leq n$, the matrix $C$ resulting from the operation of multiplication of matrices $A$ and $B$, $C = A \times B$, is such that each of its elements is denoted $ij$ with $1 \leq i \leq m$ and $1 \leq j \leq n$, and is calculated as follows

$$c_{ij} = \sum_{k=1}^{r} a_{ik}b_{kj}$$

The number of required operations to multiply $A \times B$ is $mn(2r - 1)$. 
So What Is Parallelization

The idea of parallelization is that we have $p$ parallel processors with dense matrices (not a lot of 0s). So suppose we have two square matrices $A$ and $B$ of size $n$. 

1. Partition the matrices into $p$ square blocks, where $p$ is the number of processes available.
2. Create a matrix of processes of size $p \times p$ so that each process can maintain a block of matrix $A$ and a block of matrix $B$.
3. Each block is sent to each process, and the copied sub-blocks are multiplied together and the results are added to the partial results in the $C$ sub-blocks.
4. The $A$ sub-blocks are rolled one step to the left and the $B$ sub-blocks are rolled one step upward.
5. Repeat steps 3 and 4 $\sqrt{p}$ times.
So What Is Parallelization

The idea of parallelization is that we have $p$ parallel processors with dense matrices (not a lot of 0s). So suppose we have two square matrices $A$ and $B$ of size $n$.

Here is our process:

1. Partition the matrices into $p$ square blocks, where $p$ is the number of processes available.
So What Is Parallelization

The idea of parallelization is that we have $p$ parallel processors with dense matrices (not a lot of 0s). So suppose we have two square matrices $A$ and $B$ of size $n$.

Here is our process:

1. Partition the matrices into $p$ square blocks, where $p$ is the number of processes available.
2. Create a matrix of processes of size $p^{\frac{1}{2}} \times p^{\frac{1}{2}}$ so that each process can maintain a block of matrix $A$ and a block of matrix $B$. 
So What Is Parallelization

The idea of parallelization is that we have $p$ parallel processors with dense matrices (not a lot of 0s). So suppose we have two square matrices $A$ and $B$ of size $n$.

Here is our process:

1. Partition the matrices into $p$ square blocks, where $p$ is the number of processes available.
2. Create a matrix of processes of size $p^{\frac{1}{2}} \times p^{\frac{1}{2}}$ so that each process can maintain a block of matrix $A$ and a block of matrix $B$.
3. Each block is sent to each process, and the copied sub-blocks are multiplied together and the results are added to the partial results in the $C$ sub-blocks.
So What Is Parallelization

The idea of parallelization is that we have $p$ parallel processors with dense matrices (not a lot of 0s). So suppose we have two square matrices $A$ and $B$ of size $n$.

Here is our process:

1. Partition the matrices into $p$ square blocks, where $p$ is the number of processes available.
2. Create a matrix of processes of size $p^{\frac{1}{2}} \times p^{\frac{1}{2}}$ so that each process can maintain a block of matrix $A$ and a block of matrix $B$.
3. Each block is sent to each process, and the copied sub-blocks are multiplied together and the results are added to the partial results in the $C$ sub-blocks.
4. The $A$ sub-blocks are rolled one step to the left and the $B$ sub-blocks are rolled one step upward.

$\sqrt{p}$ times.
So What Is Parallelization

The idea of parallelization is that we have $p$ parallel processors with dense matrices (not a lot of 0s). So suppose we have two square matrices $A$ and $B$ of size $n$.

Here is our process:

1. Partition the matrices into $p$ square blocks, where $p$ is the number of processes available.
2. Create a matrix of processes of size $p^{\frac{1}{2}} \times p^{\frac{1}{2}}$ so that each process can maintain a block of matrix $A$ and a block of matrix $B$.
3. Each block is sent to each process, and the copied sub-blocks are multiplied together and the results are added to the partial results in the $C$ sub-blocks.
4. The $A$ sub-blocks are rolled one step to the left and the $B$ sub-blocks are rolled one step upward.
5. Repeat steps 3 and 4 $\sqrt{p}$ times.
Parallelization Example

Example
Multiply

\[ A = \begin{bmatrix}
2 & 1 & 5 & 3 \\
0 & 7 & 1 & 6 \\
9 & 2 & 4 & 4 \\
3 & 6 & 7 & 2
\end{bmatrix} \times B = \begin{bmatrix}
6 & 1 & 2 & 3 \\
4 & 5 & 6 & 5 \\
1 & 9 & 8 & -8 \\
4 & 0 & -8 & 5
\end{bmatrix} \]
Divide the matrices into 4 square blocks as follows:

\[
\begin{array}{cc|cc}
P_{0,0} & P_{0,1} & P_{0,0} & P_{0,1} \\
2 & 1 & 5 & 3 \\
0 & 7 & 1 & 6 \\
\end{array}
\begin{array}{cc|cc}
 & & & \\
P_{1,0} & P_{1,1} & P_{1,0} & P_{1,1} \\
9 & 2 & 4 & 4 \\
5 & 3 & 7 & 2 \\
\end{array}
\begin{array}{cc|cc}
 & & & \\
 & & & \\
6 & 1 & 2 & 3 \\
4 & 5 & 6 & 5 \\
1 & 9 & 8 & -8 \\
4 & 0 & -8 & 5 \\
\end{array}
\]
Parallelization Example

Divide the matrices into 4 square blocks as follows:

\[
\begin{array}{cc}
P_{0,0} & P_{0,1} \\
2 & 1 & 5 & 3 \\
0 & 7 & 1 & 6 \\
\end{array}
\begin{array}{cc}
P_{0,0} & P_{0,1} \\
6 & 1 & 2 & 3 \\
4 & 5 & 6 & 5 \\
\end{array}
\begin{array}{cc}
P_{1,0} & P_{1,1} \\
9 & 2 & 4 & 4 \\
5 & 3 & 7 & 2 \\
\end{array}
\begin{array}{cc}
P_{1,0} & P_{1,1} \\
1 & 9 & 8 & -8 \\
4 & 0 & -8 & 5 \\
\end{array}
\]

Now, after the initial alignment, \(A\) and \(B\) become

\[
\begin{array}{cccc}
2 & 1 & 5 & 3 \\
0 & 7 & 1 & 6 \\
4 & 4 & 9 & 2 \\
7 & 2 & 5 & 3 \\
\end{array}
\begin{array}{cccc}
6 & 1 & 8 & -8 \\
4 & 5 & -8 & 5 \\
1 & 9 & 2 & 3 \\
4 & 0 & 6 & 5 \\
\end{array}
\]
Parallelization Example

So, we get the following:

\[
C_{0,0} = \begin{bmatrix} 2 & 1 \\ 0 & 7 \end{bmatrix} \times \begin{bmatrix} 6 & 1 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 16 & 7 \\ 28 & 35 \end{bmatrix}
\]

\[
C_{0,1} = \begin{bmatrix} 5 & 3 \\ 1 & 6 \end{bmatrix} \times \begin{bmatrix} 8 & -8 \\ -8 & 5 \end{bmatrix} = \begin{bmatrix} 16 & -25 \\ -40 & 22 \end{bmatrix}
\]

\[
C_{1,0} = \begin{bmatrix} 4 & 4 \\ 7 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 9 \\ 4 & 0 \end{bmatrix} = \begin{bmatrix} 20 & 36 \\ 15 & 63 \end{bmatrix}
\]

\[
C_{1,1} = \begin{bmatrix} 9 & 2 \\ 5 & 3 \end{bmatrix} \times \begin{bmatrix} 2 & 3 \\ 6 & 5 \end{bmatrix} = \begin{bmatrix} 30 & 37 \\ 42 & 39 \end{bmatrix}
\]
Now, shift $A$ one step to the left and $B$ one step up, rotating appropriately within the matrix.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>-8</td>
</tr>
</tbody>
</table>
Now the local matrix multiplication.

\[
\begin{align*}
C_{0,0} &= \begin{bmatrix} 2 & 1 \\ 0 & 7 \end{bmatrix} \times \begin{bmatrix} 6 & 1 \\ 4 & 5 \end{bmatrix} \\
&= \begin{bmatrix} 16 & 7 \\ 28 & 35 \end{bmatrix} + \begin{bmatrix} 17 & 45 \\ 25 & 9 \end{bmatrix} = \begin{bmatrix} 33 & 52 \\ 53 & 44 \end{bmatrix} \\
C_{0,1} &= \begin{bmatrix} 5 & 3 \\ 1 & 6 \end{bmatrix} \times \begin{bmatrix} 8 & -8 \\ -8 & 5 \end{bmatrix} \\
&= \begin{bmatrix} 16 & -25 \\ -40 & 22 \end{bmatrix} + \begin{bmatrix} 10 & 11 \\ 42 & 35 \end{bmatrix} = \begin{bmatrix} 26 & -14 \\ 2 & 57 \end{bmatrix} \\
C_{1,0} &= \begin{bmatrix} 4 & 4 \\ 7 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 9 \\ 4 & 0 \end{bmatrix} \\
&= \begin{bmatrix} 20 & 36 \\ 15 & 63 \end{bmatrix} + \begin{bmatrix} 62 & 19 \\ 42 & 33 \end{bmatrix} = \begin{bmatrix} 82 & 55 \\ 57 & 96 \end{bmatrix} \\
C_{1,1} &= \begin{bmatrix} 9 & 2 \\ 5 & 3 \end{bmatrix} \times \begin{bmatrix} 2 & 3 \\ 6 & 5 \end{bmatrix} \\
&= \begin{bmatrix} 30 & 37 \\ 42 & 39 \end{bmatrix} + \begin{bmatrix} 0 & -12 \\ 40 & -46 \end{bmatrix} = \begin{bmatrix} 30 & 25 \\ 82 & -7 \end{bmatrix}
\end{align*}
\]
Back To Massive Parallelization

Each processor or machine simply takes a portion of the sequence to scan for smooth numbers by itself, returning the small quantity of smooth numbers that it discovers to a central processor. As soon as the central processor has accumulated enough smooth numbers, it asks all the workers to stop. In the multiple polynomial variant, it’s common to assign some of the polynomials to each machine.

One peculiar idea for massively parallelizing the sieving step, invented by Adi Shamir, is to use not computers but a specially constructed sieving device based on light emitters and sensors that he calls TWINKLE. The concept is that we have a light for each prime number whose intensity is proportional to the logarithm of that prime. Each light turns on just two times every $p$ cycles, corresponding to the two square roots of $n \ (\mod p)$. A sensor senses the combined intensity of all the lights together, and if this is close enough to the logarithm of the current value, that value is a smooth number candidate.
Each processor or machine simply takes a portion of the sequence to scan for smooth numbers by itself, returning the small quantity of smooth numbers that it discovers to a central processor. As soon as the central processor has accumulated enough smooth numbers, it asks all the workers to stop. In the multiple polynomial variant, it’s common to assign some of the polynomials to each machine.

One peculiar idea for massively parallelizing the sieving step, invented by Adi Shamir, is to use not computers but a specially constructed sieving device based on light emitters and sensors that he calls TWINKLE. The concept is that we have a light for each prime number whose intensity is proportional to the logarithm of that prime. Each light turns on just two times every $p$ cycles, corresponding to the two square roots of $n \pmod{p}$. A sensor senses the combined intensity of all the lights together, and if this is close enough to the logarithm of the current value, that value is a smooth number candidate.