# Is it Prime or not?

Ramesh Hariharan
Dept. of Computer Science and Automation
Indian Institute of Science
Bangalore 560012

If you remember my previous article, *Large Files and Small Primes*, you will recall a question which I had posed there. Suppose I give you a very large number $n$, a number which runs into a few hundred digits or more. How will you determine if the number is prime or composite? Of course, you can use a computer to do your calculations. But how will you organize these calculations so that you get the answer *fast*. Here is the method which comes to mind immediately.

| You take every number $x$ between 2 and $n-1$ and try to divide $n$ by $x$; $n$ is prime if and only if none of these divisions gives you remainder 0. How much time does this method take? |
|---|

This method requires you to perform at most $n-2$ divisions. Let $m$ denote the number of digits in $n$. Then each division requires dividing two numbers, each number having at most $m$ digits. How much time does each division take?

You will use your computer to perform the divisions, as the numbers involved are large. Suppose your computer uses the same method for dividing two natural numbers as most people do, namely the method illustrated in Fig.1[1]. Then you should be able to see that the number of *basic operations* you perform during the division is roughly of the order of $m^2$; a basic operation is one in which you add, subtract, divide, multiply or compare two single digit numbers. I say roughly $m^2$ because it could be $2m^2$ or $3m^2$, if you count exactly, but let us work with a rough estimate of $m^2$ here.

Usually, basic operations can be performed very fast on a computer and all basic operations take roughly the same amount of time. So instead of keeping track of time,

[1]Actually computers use binary arithmetic, just 0s and 1s, and not the usual 10-digit arithmetic we are used to. However, just assume that you have a computer which does decimal arithmetic instead of binary.

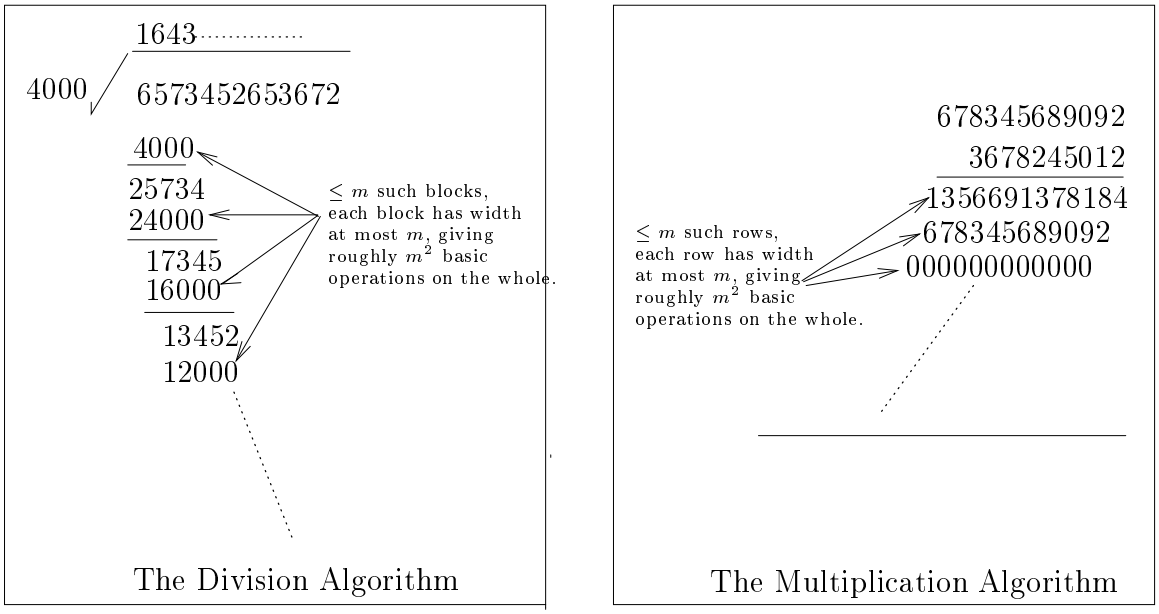


Figure 1: The Division and Multiplication Algorithms

we will keep track of the number of basic operations, and multiply by the time taken to perform a basic operation whenever we want an estimate of the time.

Thus, the number of basic operations needed for determining whether or not $n$ is prime using the above method is roughly $(n-2)m^2$. Recall that $m$ is in a few hundreds, let us say 500. Then $n$ could be as large as $10^{500}-1$ (which is the largest number you can write using 500 decimal digits?).

The fastest computers now can perform far fewer than $10^{20}$ basic operations in 1 second, which means that the above primality testing method could take time up to (approximately) $\frac{10^{500}500^2}{10^{20}}$ seconds, which if you do a quick calculation is a mind-boggling amount of time, much more than millions of millions of millions of years.

Is there another method then, a faster one? You can modify the above method slightly and try only those $x$'s which are between 2 and $\lfloor\sqrt{n}\rfloor$ and ignore those which are more than $\lfloor\sqrt{n}\rfloor$. Can you see why this is sufficient (can every factor of $n$ be larger than $\lfloor\sqrt{n}\rfloor$ if $n$ is not a prime?)? The number of basic operations for this method is roughly $\sqrt{n}m^2$, which is an improvement. But does it hold hope? The time needed is approximately $\frac{10^{250}(500^2)}{10^{20}}$, which is still unimaginably large!

By now, you should have realized that the problem term above is not $(500^2)$ but $10^{500}$ or $10^{250}$. This latter term is the entire cause of the time taken by the above methods being so large. Just $500^2$ basic operations themselves would have got done in a jiffy! If we have to test for primality in a reasonable amount of time, we have to attack this $10^{500}$ term which is *exponential* in the number of digits $m$ of the given number $n$ (that is, it has $m$ in the exponent), as opposed to the $(500)^2$ term, which is *polynomial* in $m$ (that is, it has $m$ as the base and a small exponent like 2 in this case). Thus, exponential terms are bad and polynomial terms are good. This leads to the following question.

Is there a method for checking whether or not $n$ is prime using a number of basic operations which is just polynomial in $m$? Recall $m$ is the number of digits of $n$. So we want the number of basic operations to be something like $500^2$ or $500^3$, or even $500^4$ (the time taken assuming a reasonable speed of $10^6$ basic operations per second you will get on a modern PC is then well below a day, which is not too bad in comparison).

In this article, you will find a method which *almost* does the above. I say "almost" for reasons which will be clear to you at the end. So read on.

# A Cleverer Method.

The method I have in mind is based on several fascinating properties that numbers have. In this article, let us concentrate on two such properties. These properties will *almost* (but not quite, as you will see) give us a method for checking whether $n$ is prime using just a few basic operations, polynomial in $m$, that is.

The first of these properties is called Fermat's Little Theorem.

**Fermat's Little Theorem:** Take any number $a$, calculate $a^{n-1}$, divide by $n$, and take the remainder that you get. If $n$ is a prime, then this remainder is always 1! In the language of arithmetic, this is written as

$$a^{n-1} \equiv 1 \pmod{n}$$

for all $a$ and all prime $n$.

To see how truly remarkable this property is, have your friend choose any prime number $n$ and any number $a$ and ask her to calculate $a^{n-1} \bmod n$ (that is, the remainder of $a^{n-1}$ with respect to $n$), without revealing $a$ or $n$ to you. And watch the

amazement on her face when you say that you can see what is going on in her mind and predict her answer.

How does Fermat's Little Theorem help in our current task of checking the primality of $n$? It suggests the following test:

**The Method.** Repeat the following steps several times, 20 times, say.

- Choose a random number $a$, $1 \leq a \leq n-1$.
- Calculate $a^{n-1} \bmod n$.
- Calculate $\gcd(a, n)$.

What does the above test tell us about $n$? If $n$ is a prime then each gcd and remainder obtained above must equal 1. Therefore, if any gcd/remainder obtained is not 1 then $n$ must be a composite number.

But what happens when all the gcd's and remainders obtained above are indeed 1? Can we say that $n$ is prime? Not really. Certainly, all the gcd's could be 1 even when $n$ is composite; for example, consider $n = 143 = 11 * 13$, if all the $a$'s that we choose are neither multiples of 11 nor of 13 then all our gcd's will be 1. How about the remainders? Well, Fermat's Little Theorem does not rule out composite $n$ producing remainder 1. So we cannot say that $n$ is a prime in this case. Therefore, we need a second property which will enable us to conclude something when all the gcd's and remainders come out to be 1.

Here is the second property. It is a little complicated and may take a few minutes to sink in.

**Half the $a$'s are good if one $a$ is good.** Let $S$ be the set of those numbers in the range $1 \ldots n-1$ which are relatively prime to $n$ (so each number in $S$ has gcd 1 with $n$). Call an $a$ in $S$ *good* if $a^{n-1} \not\equiv 1 \pmod{n}$. Then, if even a single $a$ in $S$ is good then at least half the $a$'s in $S$ are good.

So what does the above property say? Consider the case when all the gcd's come out to be 1. Then all the 20 $a$'s that were chosen came from $S$. The second property says that if there is a single $a$ in $S$ which is good then half the $a$'s in $S$ are good. For a prime number, no $a$ in $S$ will be good by Fermat's Little Theorem. Even a composite number need not have an $a$ in $S$ which is good. But assume for a moment that **every composite number indeed has an $a$ in $S$ which is good**. This is an important assumption to which we will get back a little later. But assume it is true

for now. With this assumption, there are several $a$'s in $S$ that are good for composite $n$. And no $a$'s in $S$ that are good for prime $n$. Now, it becomes a little clearer how the new method works.

You calculate 20 gcd's and remainders, each for a randomly chosen $a$ between 1 and $n-1$. If any one of these gcd's/remainders is not equal to 1 then you are sure that $n$ must be composite. On the other hand, if all the gcd's/remainders calculated are 1, then you cannot be sure that $n$ is prime. But you can say that it is very very likely that $n$ is prime. Why? Because, if $n$ were composite then half the $a$'s in $S$ are good and will give a remainder which is not 1 and it is very likely that in at least one of the 20 trials, one such $a$ would have been chosen and a remainder which is not 1 obtained. But this did not happen, and therefore, $n$ is likely to be a prime.

I am sure you are a little uncomfortable about the above method. You will probably have to mull over it a little bit, especially the "likely to be prime" part. So let us go into this in a little more detail. If all gcd's/remainders are 1, then $n$ is likely to be a prime, but how likely? Can we determine what are the odds that you make an error when you declare $n$ to be prime in this case? Yes, indeed, let us determine the odds.

Suppose $n$ is composite. Then what are the odds that all the 20 gcd's/remainders come out to be 1 and you declare $n$ to be prime. We know that at least half the $a$'s in $S$ will give remainders which are not 1. Then there is at most a 1/2 chance that the first gcd is 1 and the first remainder is 1 as well (this is like tossing a coin and wanting heads). The chance that the first two gcd's and remainders are both 1 is even smaller, it is 1/4 (this is like tossing a coin twice and wanting heads both times). Similarly the chance that all the 20 gcd's/remainders are 1 is at most $\frac{1}{2^{20}}$, which is minuscule, just .00000095. Thus the odds that you are making an error when declaring $n$ to be prime if all 20 gcd's/remainders turn out to be 1 are really small!

Fine, so this new method seems to give correct results, largely. Chances of error are very small. But how about the the number of basic operations needed? We need to compute $\gcd(a, n)$ and $a^{n-1} \bmod n$ 20 times. How much time does each such calculation take?

## Calculating $a^{n-1} \bmod n$

To calculate $a^{n-1} \bmod n$, at first sight, you might think that you need to perform $n-2$ multiplications+divisions[2] (you calculate $a^2 \bmod n, a^3 \bmod n, a^4 \bmod n$ and so on until you have $a^{n-1} \bmod n$, each term requires multiplying the previous term by $a$ and then dividing the result by $n$ to get the remainder). Notice that each multiplication involves two numbers having at most $m$ digits each (because each time we take a remainder, we get a number smaller than $n$ and $n$ itself has only $m$ digits).

[2] A multiplication+division is a multiplication followed by a division.

Again, assume that your computer uses exactly the same method for multiplication as the one most people would use (see Fig.1). Then each multiplication will take at most (roughly) $m^2$ basic operations. Each remainder calculation is done by dividing by $n$ and also takes roughly $m^2$ basic operations (see Fig.1). Thus, the total number of basic operations needed by our new method is $(n-2)2m^2$, which is not very different from the number for our old method. So have we made any progress at all?

Progress is made by observing that calculating $a^{n-1} \bmod n$ does not need $n-2$ multiplications and divisions, if you are clever. For example, suppose $n-1$ was $1024=2^{10}$. Then instead of calculating $a^2 \bmod n, a^3 \bmod n, \ldots, a^{1024} \bmod n$, you could use repeated squaring to calculate

| |
|---|
| $a^2 \bmod n$, $a^4 \bmod n$, $a^8 \bmod n$, $a^{16} \bmod n$, $a^{32} \bmod n$, $a^{64} \bmod n$, $a^{128} \bmod n$, $a^{256} \bmod n$, $a^{512} \bmod n$, $a^{1024} \bmod n$. |

The latter series has just 10 terms as opposed to the former which has 1023 terms. Every term in the latter series is obtained by squaring the previous term and then finding the remainder with respect to $n$. Thus only 10 multiplications+divisions are needed to calculate $a^{1024} \bmod n$.

Can you now see that if $n-1$ was a power of 2, say $2^k$, then you can calculate $a^{n-1} \bmod n$ in just $k$ multiplications+divisions? And $k = \log_2(n-1)$, which is much smaller than $n-2$. But what if $n-1$ is not a power of 2? For example, suppose $n-1 = 1021$. Then will you need to perform 1020 multiplications+divisions or can the above trick of repeated squaring still be used?

Let us see. Write 1021 as a sum of powers of 2:

$$1 + 4 + 8 + 16 + 32 + 64 + 128 + 256 + 512$$

So $a^{1021} \bmod n$ can be written as:

$$a^1 a^4 a^8 a^{16} a^{32} a^{64} a^{128} a^{256} a^{512} \bmod n$$

Next, calculate the terms of the series $a^1 \bmod n$, $a^2 \bmod n$, $a^4 \bmod n$, $a^8 \bmod n$, $a^{16} \bmod n$, $a^{32} \bmod n$, $a^{64} \bmod n$, $a^{128} \bmod n$, $a^{256} \bmod n$, $a^{512} \bmod n$, as before, but be careful not to forget the previous term of the series after you calculated the next one. What needs to be done now? All you need to do is to multiply together all the terms of the above series (except the second term), and with each multiplication do a division to calculate the remainder with respect to $n$. How many multiplications+divisions does this lead to? You need 9 multiplications+divisions to calculate the 10 terms of the series and then another 8 multiplications+divisions to combine the 9 relevant terms together, a total of just 17 multiplications+divisions!

So in general, if $n-1$ is not a power of 2 then you must write $n-1$ as a sum of powers of 2. How is this done? The method is to divide $n-1$ by 2, then divide the quotient obtained by 2, then divide this quotient by 2 and so on, until the quotient becomes 0. Write down all the remainders obtained in this process. These remainders will tell you exactly how to write $n-1$ as a sum of powers of 2. Let us just illustrate this with two examples. For $n = 10$, the remainders obtained will be $0, 1, 0, 1$; so,

$$10 = 0.2^0 + 1.2^1 + 0.2^2 + 1.2^3 = 2^1 + 2^3$$

Notice that each remainder obtained is multiplied with the appropriate power of 2 (this power is determined by the position of the remainder in the above sequence) to express $n-1$ as a sum of powers of 2. For $n = 64$, the remainders obtained will be $0, 0, 0, 0, 0, 0, 1$; so,

$$64 = 0.2^0 + 0.2^1 + 0.2^2 + 0.2^3 + 0.2^4 + 0.2^5 + 1.2^6 = 2^6$$

Let us now summarise our method for calculating $a^{n-1} \bmod n$.

**Calculating $a^{n-1} \bmod n$ fast.** First, we will write $n-1$ as a sum of powers of 2. If $2^k \leq n-1 < 2^{k+1}$ then there will be at most $k$ terms in this sum and you should be able to see that just $k$ divisions by 2 are needed in this process. Second, we will calculate $a^1 \bmod n, a^2 \bmod n, a^4 \bmod n, \ldots, a^{2^k} \bmod n$, using repeated squaring+division. This will require $k$ multiplications+divisions. Finally, we will multiply together the relevant terms taking the remainder after each multiplication; this will require at most another $k$ multiplications+divisions. Thus only a total of $3k$ multiplications+divisions are needed to calculate $a^{n-1} \bmod n$. And $3k$ is at most $3 \log_2 n$.

Thus the total number of basic operations needed to calculate $a^{n-1} \bmod n$ is at most $3 \log_2 n . 2m^2$ (a note that each multiplication and each division involves two numbers each having at most $m$ digits and takes $m^2$ basic operations, a multiplication+division will take $2m^2$ basic operations).

## Calculating $\gcd(a, n)$.

Calculating gcd's can be done with what is believed to be the oldest algorithm known to man: Euclid's algorithm. Let me illustrate this algorithm with an example here.

Suppose you want the gcd of 100 and 55. Then you find the remainder of 100 with respect to 55, so you get 45. You now have 3 numbers with you, 100, 55 and 45. Throw away the largest of these, namely 100. Keep 55 and 45. Next find the

remainder of 55 with respect to 45. You get 10. Again you have 3 numbers, 55, 45 and 10; throw away 55 and continue. The next remainder will be that of 45 with respect to 10, which is 5. Now throw away 45, you have 10 and 5 left and the next remainder is 0. You halt when you get a remainder 0; the smallest non-zero number you have obtained so far (5 in this case) is the gcd!!

I will leave it as an exercise for you to prove that Euclid's method will calculate the correct gcd. The essential reason is the following: $\gcd(a, n)$ is the same as $\gcd(n \bmod a, a)$.

Let us now count the number of basic operations required by this method. All you need to do in this method is to find a sequence of remainders, each remainder requiring one division. But how many remainders need to be calculated? To see this let us write down the sequence of numbers obtained in the above example. The question becomes: how long can this sequence be?

$$100, 55, 45, 10, 5$$

Notice that the numbers keep going down in value. In addition, did you notice a curious property: the third number is at most half the first, the fifth number is at most half the third, etc. This is true, no matter what two numbers you begin with. Again, try to show this yourself. The total length of the above sequence can therefore be at most $2\log_2 n$ (you start with the largest number being $n$, the third number is at most $n/2$, the 5th is at most $n/4$, and so on). The number of basic operations needed is thus at most $2\log_2 n.m^2$ (there are $2\log_2 n$ divisions, each requiring $m^2$ basic operations). Let us summarise Euclid's algorithm.

**Euclid's Method for Calculating** $\gcd(a, n)$**.**

Calculate the following sequence:
- $i_1 = n \bmod a$.
- $i_2 = a \bmod i_1$.
- $i_3 = i_1 \bmod i_2$.
- $i_4 = i_2 \bmod i_3$.
- $i_5 = i_3 \bmod i_4$.

and so on, until 0 is obtained. The smallest non-zero number obtained is the required gcd. The number of divisions required is at most $2\log_2 n$.

# Totalling the Number of Basic Operations.

We have now established that the number of basic operations required to calculate $\gcd(a, n)$ is at most $2\log_2 n.m^2$ and the number of basic operations required to cal-

culate $a^{n-1}$ mod $n$ is at most $3 \log_2 n.2m^2$. And our method for checking whether $n$ is prime involves doing the above calculations 20 times. So the total number of basic operations is

$$20(2 \log_2 n.m^2 + 3 \log_2 n.2m^2) = 160 \log_2 n.m^2$$

Since $n \leq 10^m - 1$, $\log_2 n \leq m \log_2 10$, and the number of basic operations is at most

$$160 \log_2 10.m^3$$

And this is polynomial in $m$, not exponential!!

You can do a quick estimate to convince yourself that this new method is much faster than the older ones. Again, suppose $n$ is a 500 digit number. The number of basic operations is thus $160. \log_2 10.500^3$. At a realistic speed of $10^6$ basic operations per second on a PC, the time taken by our new method would be about 20 hours, a substantial improvement over our earlier methods which would have taken much much more than millions of trillions of years!!

## Getting Back to the Assumption.

So what do we have at the end of all this. Recall our earlier assumption that each composite number $n$ has at least one $a$ in $S$ such that $a^{n-1}$ mod $n$ is not 1. If this is indeed true then we have a reasonably good method for checking whether $n$ is prime. And the odds of this method giving a wrong answer are extremely low. But does the above assumption hold? Unfortunately, it doesn't! The so called Carmichael numbers throw a spanner in the works.

**Carmichael Numbers.** These are composite numbers for which $a^{n-1} \equiv 1 \pmod{n}$ for all $a$ in $S$. You can check that 561 is an example of such a number.

Carmichael numbers are rare but they do exist. So all natural numbers can be divided into 3 groups. First, there are primes, for which $a^{n-1} \equiv 1 \pmod{n}$ for all $a$. Second, there are Carmichael numbers, for which also $a^{n-1} \equiv 1 \pmod{n}$ for all $a$ in $S$. Third, there are the non-Carmichael composite numbers, for which at least half the $a$'s in $S$ give $a^{n-1} \not\equiv 1 \pmod{n}$. Our method of choosing a random $a$ from $1..n-1$ and calculating $\gcd(a, n)$ and $a^{n-1}$ mod $n$, repeating this 20 times, can only differentiate between prime and Carmichael numbers on one hand and non-Carmichael composite numbers on the other.

• If some gcd is not 1 then you can be sure that $n$ is composite.
• If all gcd's are 1 but some remainder is not 1 then you can be sure that $n$ is a non-Carmichael composite number.
• But if all gcd's/remainders are 1 then you only know that $n$ is very likely to be either prime or Carmichael.

So we do not yet have a method for checking primality in which the number of basic operations is a polynomial in $m$. We have to detect Carmichael numbers somehow. In this article, we will just stop here with the proof of Fermat's Little Theorem. Handling Carmichael numbers will be the topic of a later article.

# Proof of Fermat's Little Theorem.

There are several proofs, all quite simple. Here is one based on induction. The base case is $a = 1$; then clearly, $1^{n-1} \equiv 1 \pmod{n}$. Next, as the induction hypothesis, assume that the theorem is true for $a > 1$, and let us try to show it for $a + 1$. We need to show that $(a+1)^{n-1} \equiv 1 \pmod{n}$ if $n$ is a prime. In fact, let us show that $(a+1)^n \equiv a+1 \pmod{n}$ (if $(a+1)^n$ and $a+1$ have the same remainder with respect to $n$, then so do $(a+1)^{n-1}$ and 1). Expand using the Binomial Theorem:

$$(a+1)^n \equiv a^n + \binom{n}{1}a^{n-1} + \binom{n}{2}a^{n-2} + \cdots + \binom{n}{n-1}a^1 + 1 \pmod{n}$$

We will simplify the right hand side above using the following two facts. First, note that $\binom{n}{i}$ is always divisible by $n$ if $n$ is a prime and $i \neq 0, n$. The reason for this is that when you write out $\binom{n}{i}$ as $\frac{n(n-1)\cdots(n-i+1)}{i(i-1)\cdots 1}$, $n$ appears at the top and being a prime does not get divided by any of the terms in the denominator. Second, $a^n \bmod n$ equals $a \bmod n$, by the induction hypothesis.

$$\begin{aligned}
&(a+1)^n \\
&\quad\equiv \quad a^n + \tbinom{n}{1}a^{n-1} + \tbinom{n}{2}a^{n-2} + \cdots + \tbinom{n}{n-1}a^1 + 1 \pmod{n} \\
&\quad\equiv \qquad\qquad a^n + 1 \pmod{n} \\
&\quad\equiv \qquad\qquad a + 1 \pmod{n}
\end{aligned}$$

This is exactly what we required.

# An Exercise

I haven't given the proof of the second property here, the property which says that if one $a$ in $S$ is good then half the $a$'s in $S$ are good. I will give you a hint here and ask you to try to prove this property on your own.

Let $b_1, b_2, \ldots b_r$ be those numbers in $S$ such that $b_i^{n-1} \equiv 1 \pmod{n}$, and $a$ be a number in $S$ such that $a^{n-1} \not\equiv 1 \pmod{n}$, Then can you show that:

1. $ab_1 \bmod n, ab_2 \bmod n, \ldots, ab_r \bmod n$ are all distinct (you need to use the fact that $a, b_1, b_2, \ldots, b_r$ are all in $S$ and therefore each is relatively prime to $n$.
2. $(ab_i)^{n-1} \not\equiv 1 \pmod{n}$, for all $i$ from 1 to $r$.

And from these two facts, can you infer that at least half of the numbers in $S$ are good?