Chapter 1: The Problem Shor Solves

Shor’s algorithm is famous because it changes the status of a very old problem: integer factoring. Before we discuss qubits, quantum gates, or Fourier transforms, we need to understand the problem itself.

At first glance, factoring seems ordinary. If someone gives us

$$15,$$

we can write

$$15 = 3 \times 5.$$

If someone gives us

$$91,$$

we may try a few small divisors and find

$$91 = 7 \times 13.$$

For small numbers, factoring feels like arithmetic practice. But computationally, the question is not whether we can factor small numbers by hand. The real question is:

> Given a large integer N, how many computational steps are needed to find its nontrivial factors?

A nontrivial factor of N is a divisor of N other than 1 and N itself. For example, 3 and 5 are nontrivial factors of 15. The numbers 1 and 15 are trivial factors because every positive integer N is divisible by 1 and by itself.

The integer factoring problem can be stated as follows:

> Input: A positive integer N. > Output: A decomposition of N into prime factors, or at least one nontrivial factor if N is composite.

A prime number is an integer greater than 1 whose only positive divisors are 1 and itself. Examples are

$$2, 3, 5, 7, 11, 13.$$

A composite number is an integer greater than 1 that is not prime. Examples are

$$4 = 2^2, \qquad 12 = 2^2 \times 3, \qquad 21 = 3 \times 7.$$

Factoring asks us to take a composite number and reveal the smaller numbers from which it is built.

That sounds simple. The surprise is that multiplication and factoring appear to have very different computational difficulty.

If we know

$$p = 1009, \qquad q = 1013,$$

then multiplying them is straightforward:

$$1009 \times 1013 = 1{,}022{,}117.$$

But if we are only given

$$1{,}022{,}117,$$

and asked to recover 1009 and 1013, the task is less direct. We can try division by possible factors, but for large numbers this quickly becomes impractical.

This asymmetry—easy to multiply, apparently hard to factor—is one of the ideas behind RSA public-key cryptography, introduced by Rivest, Shamir, and Adleman (1978).

Input size: why “large” means many bits

When computer scientists measure the difficulty of a problem, they do not usually measure time as a function of the number N itself. They measure time as a function of the input length.

For an integer N, the input length is approximately the number of bits needed to write N in binary. If

$$2^{n-1} \leq N < 2^n,$$

then N needs n bits in binary.

For example,

$$15 = 1111_2$$

has 4 bits, while

$$255 = 11111111_2$$

has 8 bits.

This distinction matters. A trial division algorithm might test possible divisors up to √(N). But if N has n bits, then N is roughly the size of 2^n, so

$$\sqrt{N} \approx 2^{n/2}.$$

Thus trial division takes time exponential in the number of bits. It may look reasonable as a function of N, but it is enormous as a function of the actual input length.

This is one of the first lessons of computational complexity:

> A problem is considered efficiently solvable if its running time grows like a polynomial in the input length, not like an exponential function.

A polynomial-time algorithm has running time bounded by something like

$$n^2, \quad n^3, \quad n^{10},$$

where n is the number of input bits. An exponential-time algorithm has running time like

$$2^n, \quad 3^n, \quad 2^{n/2}.$$

The exact constants and powers matter in practice, but the broad distinction is essential. Polynomial growth is usually regarded as scalable; exponential growth eventually becomes overwhelming.

No polynomial-time classical algorithm for general integer factoring is known. The best known general-purpose classical factoring methods, such as the number field sieve, are much faster than trial division but still do not give polynomial-time factoring in the input length as currently understood (Lenstra and Lenstra, 1993). Shor’s algorithm changed the theoretical landscape by showing that an ideal quantum computer can factor integers in polynomial time (Shor, 1997).

Why factoring matters for cryptography

Factoring is not only a pure mathematical challenge. It matters because modern digital security often depends on problems that are easy in one direction and hard in the reverse direction.

A public-key cryptosystem is a cryptographic system with two different keys:

- a public key, which anyone may know;
- a private key, which must be kept secret.

The public key may allow anyone to encrypt a message or verify a signature. The private key allows the intended owner to decrypt or create signatures.

RSA is the classic example. In RSA, one begins by choosing two large primes p and q, then multiplying them:

$$N = pq.$$

The number N becomes part of the public key. The primes p and q remain secret. The security of RSA is closely connected to the difficulty of recovering p and q from N. If an attacker can factor N, then the attacker can compute secret information needed to break the RSA system in its standard mathematical form (Rivest, Shamir, and Adleman, 1978; Menezes, van Oorschot, and Vanstone, 1996).

For a tiny example, suppose the public modulus is

$$N = 55.$$

Factoring gives

$$55 = 5 \times 11.$$

For such a small N, this is immediate. Real RSA moduli are hundreds or thousands of bits long, so the corresponding factors are far beyond hand calculation and beyond naive search.

The cryptographic idea is not that factoring is impossible. It is that factoring carefully chosen large numbers is believed to be infeasible for classical computers at sufficiently large key sizes. Shor's algorithm says that this belief would no longer protect RSA against a large enough fault-tolerant quantum computer (Shor, 1997).

This does not mean that all cryptography is broken by Shor's algorithm. Shor's algorithm specifically threatens systems whose security depends on integer factoring or discrete logarithms. Later in the book, we will return to this distinction when we discuss cryptographic consequences.

Factoring is not primality testing

Factoring is easy to confuse with primality testing, but they are different computational problems.

Primality testing asks:

> Given N, is N prime or composite?

Factoring asks:

> Given composite N, what are its factors?

For example, take

$$N = 91.$$

A primality test only needs to answer:

$$91 \text{ is composite.}$$

A factoring algorithm must give more information:

$$91 = 7 \times 13.$$

So factoring is a search problem: it asks us to find hidden structure. Primality testing is a decision problem: it asks for a yes-or-no answer.

This distinction is important because primality testing is known to be solvable in deterministic polynomial time. The AKS primality test, introduced by Agrawal, Kayal, and Saxena, proved that the problem PRIMES is in P (Agrawal, Kayal, and Saxena, 2004). In simpler language: there is a classical algorithm that can determine whether a number is prime in polynomial time.

But knowing that a number is composite does not automatically give its factors. A primality test might tell us:

$$N \text{ is not prime},$$

while still leaving open the harder question:

$$N = ? \times ?$$

For RSA, this difference matters. If everyone can check that a number has the right general form, that does not mean everyone can recover its secret prime factors.

Factoring is also not the same as discrete logarithms

Another problem often mentioned together with factoring is the discrete logarithm problem. Shor's original quantum algorithm handles both factoring and discrete logarithms in polynomial time (Shor, 1997), but the two problems are not the same.

To understand discrete logarithms, first recall ordinary exponentiation. If we compute

$$3^4 = 81,$$

then 3 is the base, 4 is the exponent, and 81 is the result.

A logarithm reverses exponentiation. Over the real numbers, saying

$$\log_3(81) = 4$$

means

$$3^4 = 81.$$

A discrete logarithm is a similar idea, but in a finite arithmetic system, often modular arithmetic.

In modular arithmetic, we work with remainders. For example,

$$20 \equiv 3 \pmod{17}$$

because 20 and 3 have the same remainder when divided by 17.

Now consider powers of 3 modulo 17:

$$3^1 \equiv 3 \pmod{17},$$

$$3^2 = 9 \equiv 9 \pmod{17},$$

$$3^3 = 27 \equiv 10 \pmod{17},$$

$$3^4 = 81 \equiv 13 \pmod{17}.$$

So if someone asks for x such that

$$3^x \equiv 13 \pmod{17},$$

one answer is

$$x = 4.$$

That is a discrete logarithm.

The general discrete logarithm problem is:

> Given a finite group, an element g, and another element h, find an exponent x such that >

$$> g^x = h. >$$

A group is a mathematical system where elements can be combined by an operation that behaves like addition or multiplication: the operation is associative, there is an identity element, every element has an inverse, and the operation stays inside the set. We will define groups more carefully later. For now, modular multiplication is enough intuition.

Discrete logarithms are central to Diffie-Hellman key exchange and many related public-key systems (Diffie and Hellman, 1976; Menezes, van Oorschot, and Vanstone, 1996). Factoring and discrete logarithms are different problems, but Shor's algorithm attacks both because both can be transformed into problems about hidden periodic structure.

That phrase—hidden periodic structure—is the bridge from number theory to quantum computation.

The hidden pattern behind factoring

Shor's factoring algorithm does not factor N directly. Instead, it reduces factoring to another problem called order finding.

Let N be the integer we want to factor. Choose an integer a with

$$1 < a < N.$$

We are especially interested in the case where a and N have no common factor other than 1. In that case, we say a is coprime to N, meaning

$$\gcd(a, N) = 1.$$

Here gcd(a,N) means the greatest common divisor of a and N: the largest positive integer that divides both.

For example,

$$\gcd(2, 15) = 1,$$

so 2 and 15 are coprime. But

$$\gcd(5, 15) = 5,$$

so 5 and 15 are not coprime.

Now define a function

$$f(x) = a^x \bmod N.$$

This function takes an exponent x, computes a^x, and keeps only the remainder after division by N.

Let us use

$$N = 15, \qquad a = 2.$$

Then

$$f(x) = 2^x \bmod 15.$$

Compute the first few values:

$$f(0) = 2^0 \bmod 15 = 1,$$

$$f(1) = 2^1 \bmod 15 = 2,$$

$$f(2) = 2^2 \bmod 15 = 4,$$

$$f(3) = 2^3 \bmod 15 = 8,$$

$$f(4) = 2^4 \bmod 15 = 16 \bmod 15 = 1.$$

Now the pattern repeats:

$$1, 2, 4, 8, 1, 2, 4, 8, \ldots$$

The period is

$$r = 4.$$

A period of a function is a positive number r such that shifting the input by r gives the same output:

$$f(x + r) = f(x)$$

for all relevant x. The smallest positive such r is usually called the period. In number theory, for the function

$$f(x) = a^x \bmod N,$$

this period is called the order of a modulo N.

So for a=2 modulo 15, the order is

$$r = 4.$$

Why does this help factor 15? Because if we know r, sometimes we can use it to create a multiple of N that splits into two factors.

Since r=4, we know

$$2^4 \equiv 1 \pmod{15}.$$

Therefore,

$$2^4 - 1 \equiv 0 \pmod{15}.$$

So 15 divides

$$2^4 - 1.$$

Now factor the difference of squares:

$$2^4 - 1 = (2^2 - 1)(2^2 + 1).$$

That is,

$$16 - 1 = 15 = 3 \times 5.$$

The factors appear through

$$2^{r/2} - 1 = 2^2 - 1 = 3,$$

and

$$2^{r/2} + 1 = 2^2 + 1 = 5.$$

Then

$$\gcd(3, 15) = 3,$$

$$\gcd(5, 15) = 5.$$

We have recovered the factors of 15.

This example is small, but it contains the core idea:

> Factoring can be reduced to finding the period of the modular exponentiation function >

$$> f(x) = a^x \bmod N. >$$

The reduction does not work for every choice of a in one try. Sometimes a immediately shares a factor with N, which is actually good because gcd(a,N) already gives a factor. Sometimes the period r is odd, or the arithmetic produces only trivial information. But with random choices of a, repeated trials succeed with high probability. We will prove the details later.

For now, the important point is that Shor's algorithm turns factoring into period finding.

Why period finding is hard classically

The function

$$f(x) = a^x \bmod N$$

may have a period r, but the period is not usually visible from one or two values.

For example, suppose a function begins like this:

$$1, 7, 4, 13, 16, 10, 1, 7, 4, 13, 16, 10, \ldots$$

If we are only shown the first three values,

$$1, 7, 4,$$

we cannot know the period yet. It might be 3, or 6, or 100, depending on what comes next. Periods are global properties: they describe how values repeat over an extended range.

Classically, to find a period, one usually needs enough information about the function's values to detect repetition. There are clever classical algorithms for modular arithmetic and factoring, but no known classical polynomial-time algorithm for general factoring.

The quantum idea is different. Shor's algorithm does not merely sample function values one by one and wait for a repeated output. Instead, it prepares a quantum state whose amplitudes contain many inputs at once, computes the periodic function coherently, and then uses interference to make information about the period visible in a measurement.

This needs careful interpretation. It is tempting to say:

> A quantum computer tries all inputs at once.

That sentence is incomplete and often misleading. A quantum computer can create a superposition, a state described by amplitudes over many possible inputs. But when we measure, we do not get all the values. We get one outcome. The power comes from arranging the computation so that wrong possibilities cancel by destructive interference and useful global structure is amplified by constructive interference.

Shor's algorithm uses a quantum operation called the quantum Fourier transform to turn periodicity into measurable frequency information. We will study that transformation in detail later. At this point, it is enough to know the high-level shape:

1. Build a quantum state spread over many exponents x.
2. Compute a^x bmod N without destroying the quantum coherence.
3. Use interference to reveal information related to the period r.
4. Use classical number theory to convert that information into factors of N.

The algorithm is partly quantum and partly classical. The quantum part finds information about a period. The classical part checks candidates, computes greatest common divisors, and extracts factors.

The promise of Shor's algorithm

Shor's algorithm does not say that quantum computers are faster for every problem. It says something more precise and more valuable:

> Some problems contain hidden algebraic structure, and quantum mechanics can reveal that structure through interference more efficiently than known classical methods.

For factoring, the hidden structure is the period of

$$f(x) = a^x \bmod N.$$

For discrete logarithms, the hidden structure is different but related: it is also a periodic structure inside a finite algebraic system. Shor showed that both factoring and discrete logarithms can be solved in polynomial time on a quantum computer (Shor, 1997).

This is why Shor's algorithm is not just a factoring trick. It is a model for quantum algorithm design.

The transferable pattern is:

> Encode a mathematical problem as a hidden structure problem, use a quantum process to expose global periodic information, then finish with classical postprocessing.

This book will build that pattern slowly. We will begin with classical computation and reversible circuits, because quantum operations must be reversible. Then we will introduce qubits using linear algebra. After that, we will study gates, superposition, interference, entanglement, number theory, Fourier transforms, phase estimation, and modular exponentiation. Only then will the full algorithm feel natural.

For now, remember the central path:

$$\text{factoring} \longrightarrow \text{order finding} \longrightarrow \text{period finding} \longrightarrow \text{quantum Fourier sampling}.$$

The rest of the book explains each arrow.

Chapter summary

Integer factoring asks us to decompose a composite integer N into nontrivial factors. Multiplication is easy, but no polynomial-time classical algorithm is known for factoring arbitrary large integers. This apparent difficulty supports cryptographic systems such as RSA.

Factoring is different from primality testing. Primality testing asks whether N is prime or composite, and it is known to be solvable in polynomial time. Factoring asks for the actual factors.

Factoring is also different from the discrete logarithm problem, although both are important in public-key cryptography and both are solved efficiently by Shor's quantum algorithm.

The key move in Shor's factoring algorithm is to reduce factoring to order finding. Given N and a suitable a, we study

$$f(x) = a^x \bmod N.$$

This function is periodic. Its period, called the order of a modulo N, can often be used to recover factors of N. Shor's algorithm uses quantum interference, especially through the quantum Fourier transform, to find information about this period efficiently.

The problem Shor solves is therefore not only "factor this number." At a deeper level, it is:

> Find hidden periodic structure in a mathematical function, then use that structure to solve the original problem.

That idea will guide everything that follows.

References

Agrawal, M., Kayal, N., and Saxena, N. (2004). “PRIMES is in P.” Annals of Mathematics, 160(2), 781–793.

Diffie, W., and Hellman, M. E. (1976). “New Directions in Cryptography.” IEEE Transactions on Information Theory, 22(6), 644–654.

Lenstra, A. K., and Lenstra, H. W., Jr., eds. (1993). The Development of the Number Field Sieve. Lecture Notes in Mathematics, Vol. 1554. Springer.

Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. (1996). Handbook of Applied Cryptography. CRC Press.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.” Communications of the ACM, 21(2), 120–126.

Shor, P. W. (1997). “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer.” SIAM Journal on Computing, 26(5), 1484–1509.

Document information

Chapter 1: The Problem Shor Solves

Project	Shor's Algorithm from First Principles
Document	Document 1.5
Author	mujirin
Verifier	Not verified
Downloaded	July 04, 2026 18:10 KST
Status	Working
Document link	https://www.theorytrace.com/projects/shors-algorithm-from-first-principles/documents/-chapter-1-the-problem-shor-solves/