Integer Representations
The Decimal System

When we write integers, we use the **decimal system** by default. For example, when we write $n = 37612$, we mean

$$n = 3 \cdot 10^4 + 7 \cdot 10^3 + 6 \cdot 10^2 + 1 \cdot 10^1 + 2 \cdot 10^0.$$ 

Inspired by this example, we recognize the definition of a **decimal representation of a non-negative integer**: it is a representation of the integer as a sum of multiples of powers of ten, wherein each multiplier, called a **decimal digit**, is an integer in the range 0 to 9.

Formally, if $n, m$ are non-negative integers and the numbers $d_k, k = 0 \ldots m$, called digits, are integers with $0 \leq d_k < 10$, and if

$$n = \sum_{k=0}^{m} d_k 10^k$$

then $n = d_md_{m-1} \ldots d_1d_0$ is called the decimal expansion or decimal representation of $n$. Every non-negative integer $n$ has a unique decimal expansion.
Base b Expansions of Integers

There is nothing special about the number 10, except that humans have ten fingers. It therefore stands to reason that all integers should be representable in a similar fashion using powers of any other positive integer $b > 1$. This is correct.

If $b$ is a positive integer and $n, m$ are non-negative integers and the numbers $d_k, k = 0 \ldots m$, called base $b$ digits, are integers with $0 \leq d_k < b$, and if

$$n = \sum_{k=0}^{m} d_k b^k$$

then $n = (d_md_{m-1} \ldots d_1d_0)_b$ is called the base $b$ expansion or base $b$ representation of $n$. It is unique for every non-negative integer $n$ as long as we don’t allow leading zeros, i.e. $d_m \neq 0$.

Base 10 expansions are decimal expansions. Base 2 expansions are called binary expansions, base 3 ternary, base 8 octal, base 12 duodecimal, base 16 hexadecimal.

If the base $b$ is greater than 10, we need symbols that represent values of the digits that are greater than 9. In duodecimal, we use A for the value 10 and B for the value 11. In hexadecimal, we additionally use C for the value 12, D for the value 13, E for the value 14 and F for the value 15.
Examples of Base $b$ Expansions

$(10111)_2 = 2^4 + 2^2 + 2^1 + 2^0 = 23$

$(1201)_3 = 3^3 + 2 \cdot 3^2 + 3^0 = 46$

$(357)_8 = 3 \cdot 8^2 + 5 \cdot 8^1 + 7 \cdot 8^0 = 239$

$(AB34)_{12} = 10 \cdot 12^3 + 11 \cdot 12^2 + 3 \cdot 12^1 + 4 \cdot 12^0 = 18904$

$(1A2F)_{16} = 16^3 + 10 \cdot 16^2 + 2 \cdot 16^1 + 15 \cdot 16^0 = 6703$
Adding Numbers in Base $b$

Let us consider how to add two numbers given n base $b$ where $b$ is not 10. If you think that this should be done by converting the numbers into decimal, adding them, and converting the sum back into base $b$, then you are stuck in a mindset of decimal superiority.

There is nothing special about decimal. If you can carry out addition in decimal, you can carry it out in any other base, using the same algorithm you learned in elementary school. You write the two numbers underneath each other so that the last digits are vertically aligned, and then, going from right to left, you add the digits. If there is overflow, i.e. two digits have a sum of $b$ or more, then you reduce them mod $b$ and carry a 1 into the next column to the left. In the following examples, resulting digits in a column where overflow occurred are marked red.

<table>
<thead>
<tr>
<th>Binary Addition</th>
<th>Octal Addition</th>
<th>Hex Addition</th>
</tr>
</thead>
<tbody>
<tr>
<td>11001111</td>
<td>41571521</td>
<td>584ADBFF</td>
</tr>
<tr>
<td>+ 100110</td>
<td>+ 137542336</td>
<td>+ EC07A0</td>
</tr>
<tr>
<td>= 11110101</td>
<td>= 201334057</td>
<td>= 5936E39F</td>
</tr>
</tbody>
</table>
The size of a positive integer vs the number of its base b digits (I)

Let us explore the relationship between a positive integer and its number of digits with a base 10 example first.

A three-digit decimal number is at least 100, and at most 999. Thus, if \( n \) is such a number, it satisfies the inequality

\[
10^2 \leq n < 10^3.
\]

Similarly, if \( n \) has \( k \) decimal digits, then \( 10^{k-1} \leq n < 10^k \).

This leads us to the following theorem: if \( n \) has \( k \) digits in base \( b \), then

\[
b^{k-1} \leq n < b^k.
\]

This bounds are optimal. The lowest base \( b \) number with \( k \) digits is \( b^{k-1} \), the largest is \( b^k - 1 \). For example, the lowest base 7 number with 5 digits is \((10000)_7 = 7^4 = 2401\), the highest is \((66666)_5 = 7^5 - 1 = 16806\).
The size of a positive integer vs the number of its base-b digits (II)

Let us prove this theorem. Suppose $n$ is a positive integer that has $k$ digits in base $b$: $n = (d_{k-1}d_{k-2} \ldots d_1d_0)_b$ with $d_{k-1} \geq 1$. By definition, this means

$$n = \sum_{i=0}^{k-1} d_i b^i.$$

Since $d_{k-1} \geq 1$ and all other $d_i$'s are nonnegative, $n \geq d_{k-1} b^{k-1} \geq b^{k-1}$. Thus $n \geq b^{k-1}$, which is the first part of our inequality.

We also know that each digit $d_i$ is at most $b - 1$. This means

$$n \leq \sum_{i=0}^{k-1} (b - 1)b^i \leq (b - 1) \sum_{i=0}^{k-1} b^i = (b - 1) \frac{b^k - 1}{b - 1} = b^k - 1 < b^k.$$

We used the geometric summation formula here. Thus $n < b^k$, which is the second part of our inequality.
The size of a positive integer vs the number of its base b digits (III)

Knowing now that a base b number \( n \) with \( k \) digits satisfies

\[
b^{k-1} \leq n < b^k,
\]

we seek to express \( k \) in terms of \( n \). We apply the base b logarithm to the inequality and get \( k - 1 \leq \log_b n < k \).

Since the real number \( \log_b n \) is in the integer interval \( [k - 1, k) \), we can find \( k \) by flooring and adding 1:

\[
k = \lfloor \log_b n \rfloor + 1.
\]

(If \( \log_b n \) is not an integer, which will be true in most cases, instead of flooring and adding 1, we can just use the ceiling instead.)

Example: 79999 has exactly \( \lfloor \log_{16} 79999 \rfloor = \lfloor 4.07192.. \rfloor = 5 \) digits in hexadecimal. Indeed, 79999 = (1387F)\(_{16}\).
The connection to the geometric summation formula

Our proof that a positive integer $n$ that has $k$ digits in base $b$ is at most $b^k - 1$, used the geometric summation formula:

$$\sum_{i=0}^{k-1} b^i = \frac{b^k - 1}{b - 1} \quad \text{or equivalently,} \quad \sum_{i=0}^{k-1} (b - 1)b^i = b^k - 1.$$

Base $b$ representation allows us to see this formula in a different light. The left side is simply the base $b$ number with $k$ digits, all of them $b - 1$, which is the highest digit in base $b$. The right side, $b^k - 1$, is one less than the base $b$ number that starts with 1, followed by $k$ zeros.

So the geometric summation formula, for integer $b > 1$, only expresses a relationship we take for granted in the decimal system: when you subtract one from the lowest base $b$ number with $k + 1$ digits, which is a 1 followed by all zeros, you get the highest base $b$ number with $k$ digits, which consists solely of the highest digit.

A decimal example is $100000 - 1 = 99999$. Here are examples in other bases:

Binary: $10000000 - 1 = 1111111$
Hexadecimal: $10000 - 1 = FFFF$
Octal: $100 - 1 = 666$
How to compute base $b$ expansions

Let us return to our first example $n = 37612$. By applying the division algorithm with $d = 10$, we get $n = 3761 \cdot 10 + 2$. The remainder is the last digit. Then we apply the division algorithm to the quotient 3761. Its last digit is again the remainder. Repeat.

This inspires the following algorithm for finding the base $b$ digits of an integer $n$. Apply the division algorithm repeatedly as follows:

$$
n = q_1 b + r_1
$$
$$
q_1 = q_2 b + r_2
$$
$$
q_2 = q_3 b + r_3
$$

The algorithm stops as soon as the first quotient $q_n$ is zero, because that means we have reduced $n$ to its first digit. Then, the remainders $r_1, r_2, \ldots r_n$ are the base $b$ digits of $n$, from last to first.

As an example, let us apply this algorithm to convert $n = 17101$ to hexadecimal:

$$
17101 = 1068 \cdot 16 + 13 \qquad \textcolor{red}{= D}
$$
$$
1068 = 66 \cdot 16 + 12 \qquad \textcolor{red}{= C}
$$
$$
66 = 4 \cdot 16 + 2
$$
$$
4 = 0 \cdot 16 + 4
$$

Therefore, $17101 = (42CD)_{16}$
Why We Use Binary and Hexadecimal

The binary system plays a preeminent role in modern technology. A binary digit, called a bit, can only assume the values 0 and 1. This means that a technological system that stores data in binary form only needs to distinguish between two states of a memory cell, on or off, black or white, current or no current. The fact that there are no shades of gray there makes binary systems robust—circuit noise, manufacturing tolerances or degradation are generally not going to turn an on signal into an off or vice versa.

Using base $b$ representations with the lowest $b$ value possible, $b = 2$, has disadvantages as well. Among all base $b$ expansions, the binary ones have the greatest number of digits. This makes working with binary data on a computer uncomfortable for people. An alternative is hexadecimal.

The hexadecimal system enjoys a close relationship with the binary system because the hex base 16 is a power of the binary base 2: $2^4 = 16$. A direct consequence of that (which you might prove as an exercise) is that if you divide a binary number into blocks of 4 binary digits, if necessary with some zero padding on the left, each block represents one hex digit. Example:

$$(1011\ 1111\ 0001\ 0101)_2 = (BF15)_{16}$$

It only takes 4 keystrokes to type this number into a computer using hexadecimal, but it takes 16 keystrokes to do it in binary.

This kind of block conversion is also possible from octal to binary.

Since 10 is not a power of 2, such a block relationship between binary and decimal digits does not exist.
In the 80s, home computer magazines would distribute free applications and utilities to their readers as hex machine code printed in the pages of the magazine, which had to be typed in by the reader.
Base64

There are very old internet services such as email and usenet whose protocols were designed only for the transmission of text. Without going into technical details, with these protocols, it is safe to assume only that one can transmit the 10 digits [0-9], 26 uppercase letters [A-Z], 26 lowercase letters [a-z] and a small number of special characters such as +-._!:;&$%#*()@^/.

To transmit binary files over such systems efficiently, one needs to choose a base that takes maximum advantage of the available characters, which become digits. Using decimal or even hexadecimal text would be inefficient.

An important constraint is that the base should be a power of 2. That is the only way we can efficiently block-convert to and from binary.

The fact that there are 52 letters and 10 digits suggests using base 64, where [0-9], [A-Z], [a-z] account for the first 62 digits, and two “safe” special characters for digits 63 and 64, such as + and /.

There are many different ways of choosing the two extra symbols, which has lead to a whole family of Base64 encoding schemes.

Consider the following questions as an exercise: by what factor is a binary file that is saved as an ASCII text string consisting of only 0’s and 1’s shortened by encoding it in Base64? How does Base64 compare to an ASCII text with a hex representation?
Multi-Level Cell (MLC) Flash Memory

Usually, digital memory technology saves one binary digit (a bit) in each memory cell. Each cell has two states. In an effort to increase the storage capacity of flash memory, manufacturers have been increasing the number of possible states for each memory cell.

There are triple level cells (TLC) that can distinguish 8 different internal states and therefore store one octal digit, which is equivalent to 3 bits.

To store $n$ bits, a cell needs to be able to distinguish $2^n$ internal states. Since the reliability of the storage device decreases with the number of states cells need to distinguish, and since the number of states grows exponentially with the number of bits stored per cell, increasing $n$ and therefore storage capacity in this fashion becomes quickly impractical.
Efficient Binary Multiplication Through Addition and Bit Shifting (1)

Some old CPUs like the famous 8 bit MOS Technology 6502 do not have multiplication of binary numbers in their instruction set. They could however add binary numbers, and perform left bit shifts. [A left bit shift moves every bit one position to the left and appends a zero at the end: $11101 \rightarrow 111010$. ]

This raises the question how one might efficiently implement multiplication of binary numbers $n, m$ in this situation.

One obvious idea is to implement multiplication through repeated addition – we compute $nm$ by adding $n$ to itself $m - 1$ times. This approach requires $m - 1$ additions and is inefficient if $m$ is large. Let us call this “slow multiplication”.

A more efficient approach presents itself when we realize that left bit shifting, even though it is a very ‘cheap’ operation, is multiplication by 2. Thus, multiplication by $m = 2^k$ for some positive integer $k$ can be efficiently performed simply through $k$ successive left bit shifts of $n$.

Of course, most positive integers $m$ are not powers of 2. They are, however, all sums of powers of 2, and their binary representation gives us their precise decomposition into such a sum for free.

For example, having $m = (10110)_2$ tells us that $m = 2^4 + 2^2 + 2^1$. 
Efficient Binary Multiplication Through Addition and Bit Shifting (2)

We can therefore perform efficient (‘fast’) multiplication by $m = 2^4 + 2^2 + 2^{1}$ based on the following decomposition:

$$nm = 2^4n + 2^2n + 2^{1}n.$$

Computing $2^4n$ requires only 4 left bit shifts of $n$, which computes $2^2n$ and $2^{1}n$ on the way for free. Then we require an additional 2 additions.

In general, if $k$ is the number of binary digits of $m$, then the number of bit shifts required is $k - 1$, and the number of additions is no more than $k - 1$.

As we learned earlier, $k = \lfloor \log_2 m \rfloor \approx \log_2 m$.

It follows that fast multiplication by $m$ requires no more than roughly $2\log_2 m$ operations, counting both additions and shifts. This is dramatically less than the number of operations for slow multiplication by $m$, which was roughly $m$. 
Fast Multiplication is just the Binary Version of the Pencil and Paper Method

Once you work out an example of fast multiplication, you realize that it is just the old pencil and paper method of multiplication you learned in elementary school, performed in binary.

Let us carry out the binary multiplication $1010 \cdot 11011$. For each 1 bit of the multiplier 11011 in position $j$, you have to left-shift 1010 by $j$. You add all resulting numbers, making sure to carry the 1’s in case of an overflow.

For clarity, the extra zeros added by the bit shifts are marked blue. Digits in the final result where overflow occurred are marked red again.

$$1010 \cdot 11011 = 10100000$$
$$+ 1010000$$
$$+ 10100$$
$$+ 1010$$
$$= 10001110$$
The following is a pseudocode implementation of our fast multiplication algorithm.

```python
function fast_multiplication(n, m)
result = 0;
shifted_n = n;
for each bit of m (going from right to left) {
    if (bit == 1) then
        result = result + shifted_n;
        result = result + shifted_n;
    shifted_n = shifted_n << 1  // shift left by 1
}
return result;
```

A practical implementation of this algorithm would have to allocate twice as many bits for `result` as for `n` and `m`; for example, if `n` and `m` are 32-bit numbers, `result` would be a 64 bit number.
The Same Idea Applied to Exponentiation.

Just like we can exploit the binary expansion of a multiplier to dramatically speed up multiplication, we can exploit the binary expansion of an exponent to speed up exponentiation. The approach is exactly the same, the only difference is that multiplication now plays the role of addition, and squaring (raising to the power of 2) now plays the role of multiplication by 2.

In cryptography, it happens frequently that we need to compute numbers of the form $b^n \mod m$ where $n$ is very large. In that case, computing $b^n$ first and then taking its remainder is very inefficient, if not outright impossible. If $n$ is very large, then $b^n$ is such an enormous integer that either computing it or storing it would be prohibitive. Fortunately, to do so is also unnecessary.

In the previous presentation, we learned that we can reduce the factors in a product $\mod m$ before we multiply them, thereby reducing their size: $(ac) \mod m = (a \mod m)(c \mod m) \mod m$.

Since this rule can be applied repeatedly, to products containing more than two factors, we compute $b^n \mod m$ in a loop as follows:

1. Start with $a = 1$
2. For $k = 1$ to $n$ do $a = (a \cdot b) \mod m$.

When the for loop is finished, $a$ contains the value $b^n \mod m$.

This algorithm solves the problem of storage – no number greater than $m$ itself ever needs to be stored. However, it still requires $n$ multiplications and remainder operations. If, say, $n = 2^{100}$, then on a fast computer, at one trillion multiplication and remainder operations per second, the task would require over 40 billion years, three times as long as the age of the universe. We shall therefore call the algorithm above “slow modular exponentiation.”

There is a shortcut that dramatically reduces the number of multiplications required- **fast modular exponentiation**.
Fast Modular Exponentiation (I)

Let us consider the problem of computing $3^{1024}$ mod 7. Using the slow modular exponentiation algorithm from the previous slide, this would require 1024 multiplications and remainder operations. Using fast modular exponentiation, it only takes 10. The idea of the algorithm is that we get to high exponents very fast if we start with $3 \mod 7$ and keep squaring and taking remainders:

\[
3^2 \mod 7 = (3 \mod 7)^2 \mod 7 \\
3^4 \mod 7 = (3^2 \mod 7)^2 \mod 7 \\
3^8 \mod 7 = (3^4 \mod 7)^2 \mod 7 \\
3^{16} \mod 7 = (3^{8} \mod 7)^2 \mod 7 \\
\vdots \\
3^{1024} \mod 7 = (3^{512} \mod 7)^2 \mod 7
\]

This algorithm takes 10 steps because each step doubles the exponent, and $1024 = 2^{10}$.

For comparison: if the exponent $n$ was $n = 2^{100}$, then the computer described on the previous slide would require 0.1 nanoseconds (1/10 of a billionth of a second) to complete the task of computing $b^n \mod m$ using fast exponentiation, compared to 40 billion years using slow exponentiation.
Fast Modular Exponentiation (II)

For the record, let us state the general form of the fast modular exponentiation algorithm:

\[
\begin{align*}
  b^2 \mod m &= (b \mod m)^2 \mod m \\
  b^4 \mod m &= (b^2 \mod m)^2 \mod m \\
  b^8 \mod m &= (b^4 \mod m)^2 \mod m \\
  b^{16} \mod m &= (b^8 \mod m)^2 \mod m \\
  &\vdots
\end{align*}
\]

You may already have noticed that fast modular exponentiation comes with a weakness: successive squaring only produces exponents that are powers of 2. Fortunately, there is a way around that limitation, and it requires little more than binary expansion of the exponent.

Let's say we wish to compute \(3^{1035} \mod 7\). Observe that \(1035 = 1024 + 8 + 2 + 1 = 2^{10} + 2^3 + 2^1 + 2^0\). Then, using basic laws of exponents and the modular multiplication rule, we get

\[
3^{1035} \mod 7 = 3^{1024+8+2+1} \mod 7 \\
= (3^{1024} \mod 7) \cdot (3^8 \mod 7) \cdot (3^2 \mod 7) \cdot (3^1 \mod 7) \mod 7
\]

All factors needed are produced as a by-product of computing \(3^{1024} \mod 7\) using the fast modular exponentiation approach. Therefore, to find \(3^{1035} \mod 7\), all we need to do is compute \(3^{1024} \mod 7\) using fast modular exponentiation, and initialize a result variable \(a\) with 1. Each round, we check whether the current exponent \(k\) corresponds to a bit of value 1 in 1035. If so, we multiply \(a\) by the value of \(3^k \mod 7\) that we just computed, and then apply another mod 7 operation to keep the values of \(a\) small.
Fast Modular Exponentiation (III)

The following is a C/C++ implementation of fast modular exponentiation for computing $b^n \mod m$.

```c
int fast_mod_exp(int b, int n, int m) {
    int a = 1;
    int c = b;
    int p = n;
    while (p > 0) {
        if (p % 2 == 1) {
            a = a*c % m
            p = p - 1;
        }
        p = p / 2;
        c = c*c % m;
    }
    return a;
}
```

The algorithm simultaneously performs fast modular exponentiation of $b$ (with all the values of $b^{2^k} \mod m$ for $k = 0,1,2, \ldots$ stored in the variable $c$) and repeated division algorithms on $n$ to determine its bit composition, with the quotients held in the variable $p$. Every time a bit of $n$ of value $2^k$ is detected, the output variable $a$ is multiplied by the corresponding value of $c = b^{2^k} \mod m$. 
The Discrete Logarithm Problem (I)

The base-b logarithm is the inverse function of the base-b exponential function. If \( b > 1 \) and \( f : \mathbb{R} \to (0, \infty) ; \ f(x) = b^x \), \( f \) is bijective and its inverse is \( f^{-1} : (0, \infty) \to \mathbb{R} ; \ f^{-1}(x) = \log_b x \).

A discrete logarithm is the inverse of a modular exponentiation function. If \( b \) is an integer greater than 1, and

\[
f(n) = b^n \mod m
\]

then \( f \) may or may not be bijective. How to pick \( m \) and then define the domain and codomain of \( f \) to make \( f \) bijective is a topic in number theory and is beyond the scope of this class. If \( f \) is bijective, its inverse is called the base \( m \) discrete logarithm.
The Discrete Logarithm Problem (II)

Let us take $m = 11$ and $b = 2$ and study the corresponding modular exponentiation function $f(n) = 2^n \mod 11$. On the middle right is a table of values. Observe that the outputs for $0 \leq n \leq 9$ are unique, and exhaust all possible remainders mod 11 except zero. Therefore, $f$ is bijective if we consider it a function from $\{0,1, \ldots, 9\}$ to $\{1,2, \ldots, 10\}$.

Therefore, there is a discrete log function $f^{-1}$ which assigns to each integer $c$ from 1 to 10 the unique integer $n$ from 0 to 9 so that $2^n \mod 11 = c$. Its table of values is on the far right.

Observe that the values of $f^{-1}$ seem “random” – there is no formula to compute them. The only way to evaluate $f^{-1}(n)$ is to search the table of values for $f$ until we have found an output that equals $n$.

For certain choices of (very large) $m$ and $b$, the range of $f$ is very large, and not only is our search procedure impractical, but there are no known algorithms to evaluate $f^{-1}$ efficiently. This problem is known as the discrete log problem.

Functions $f$ that are invertible in principle but not in practice, with reasonable effort, are called “trap door functions”. These functions play an important role in cryptography.
Diffie-Hellman Key Exchange (I)

One of the basic problems of modern communication is how two parties who may be separated by great physical distance can enjoy privacy while communicating over an insecure channel such as the internet. Maintaining privacy requires an encryption system with a secret key that is only known to the two parties, but to agree on such a shared secret would seem to require a secure communications channel in the first place.

Diffie-Hellman key exchange, first published in 1976, resolves this apparent “chicken or the egg” dilemma.
Diffie-Hellman Key Exchange (II)

As is customary in cryptography, we shall call the two parties to the communication Alice and Bob. (Alice could be yourself, and Bob could be an internet store.) Alice and Bob wish to communicate securely over an insecure public channel such as the internet.

In Diffie-Hellman Key Exchange, both partners pick a secret number each. Alice picks $a$ and Bob picks $b$. They do not transmit those numbers to each other. These numbers will serve as exponents in a modular exponentiation.

Then they agree over the public channel on numbers $g$ and $p$. Without going into technical details, $g$ and $p$ are designed so that the discrete log problems of finding $a$, given $A = g^a \mod p$ and $b$, given $B = g^b \mod p$ are impossible to solve in practice. $p$ must be a large prime for that.

Alice computes $A$ and sends it to Bob. Bob computes $B$ and sends it to Alice. An eavesdropper ("Eve") will know $A$ and $B$, but cannot reconstruct Alice’s secret $a$ from $A$, or Bob’s secret $b$ from $B$ due to the difficulty of the discrete log problem.
Diffie-Hellman Key Exchange (III)

So far, this is unspectacular. Alice and Bob each have a secret, but they do not know each other’s secrets. All they have is what one might call a “hint” about the other’s secret: the numbers A, B. Can they combine their own secret with the hint they each have about the other’s secret to construct a shared secret? That part is the “magic” of Diffie-Hellman.

The magic lies basically in the exponential law \((x^n)^m = (x^m)^n = x^{nm}\), which also holds for modular arithmetic. If someone hands you the quantity \(x^n\), you don’t need to know what either \(x\) or \(n\) are to compute \(x^{nm}\). You just raise the given value \(x^n\) to the power of \(m\). Likewise, if someone hands you the quantity \(x^m\), you don’t need to know what either \(x\) or \(m\) are to compute \(x^{nm}\). You just raise the given value \(x^m\) to the power of \(n\).

Therefore, Alice computes \(B^a \mod p = (g^b \mod p)^a \mod p = g^{ab} \mod p = S\). Bob computes \(A^b \mod p = (g^a \mod p)^b \mod p = g^{ab} \mod p = S\).

The number \(S\) is now their shared secret, and can be used as a secret key for an encryption system. Eve cannot compute \(S\) since she neither knows \(a\) nor \(b\). She would have to solve a discrete log problem to find either.
Diffie-Hellman Key Exchange (IV)


This makes it possible for extremely-well funded adversaries such as the intelligence agencies of nation states to precompute sufficient information to solve the discrete log problem mod $p$ in reasonable time.