Prime Numbers, GCD, Euclidean Algorithm and LCM
Definition of Prime Number and Examples

A **prime** number is a positive integer that has exactly two positive divisors. Equivalently, it is an integer greater than 1 that has only 1 and itself as positive divisors. (The number 1 is not prime).

A positive integer that is greater than 1 and is not prime is called **composite**.

The first 10 prime numbers are: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29.

Observe that 2 is the only even prime. All other positive even numbers are multiples of 2 and therefore composite.
Factors and Cofactors of a Composite Integer (1)

By definition, if \( n \) is composite integer, then it must have a positive factor \( k \) other than 1 or \( n \). This factor must then satisfy \( 1 < k < n \), because a divisor of an integer cannot be greater than the integer itself. We then have the following theorem:

If \( n \) is composite and \( k \) is one of its factors such that \( 1 < k < n \), and \( m \) the cofactor \( m \) with \( n = km \), then

1. \( m \) satisfies the same inequality as \( k \): \( 1 < m < n \).
2. if \( k \leq \sqrt{n} \), then \( m \geq \sqrt{n} \).

Proof: By substituting \( k = \frac{n}{m} \) into \( 1 < k < n \), we get \( 1 < \frac{n}{m} < n \). By taking reciprocals and remembering that less than relationships between positive numbers correspond to greater than relationships between the reciprocals, we get \( \frac{1}{n} < \frac{m}{n} < 1 \). We multiply by the positive number \( n \) to get \( 1 < m < n \).

Now suppose \( k \leq \sqrt{n} \). Then if \( m < \sqrt{n} \) was true, we could multiply the inequalities to get \( n = km < \sqrt{n} \cdot \sqrt{n} = n \). \( n < n \) is a contradiction, therefore \( m < \sqrt{n} \) must have been false and \( m \geq \sqrt{n} \) must be true.
What the theorem on the previous page tells us is that the nontrivial (other than 1 and \( n \)) factor-cofactor pairs of a composite integer \( n \) lie strictly between 1 and \( n \), and each factor’s cofactor lies on the “opposite side” of the square root of \( n \), except when the square root itself is a factor, in which case it is equal to its own cofactor.

Example: the positive factors of 100 are:

\[
\begin{array}{cccccccccc}
1 & 2 & 4 & 5 & 10 & 20 & 25 & 50 & 100
\end{array}
\]
Testing for Primality

We learned that factors of a composite number $n$ come in pairs that lie on opposite sides of the square root, i.e. for each factor that is less than or equal to $\sqrt{n}$, there is one that is greater than or equal to $\sqrt{n}$ and vice versa.

Therefore, to prove that an integer $n > 1$ is a prime, we only need to test divisibility of $n$ by potential factors $k$ with $1 < k \leq \sqrt{n}$. If $n$ is divisible by none of those $k$, then $n$ is prime.

If $n$ is divisible by a composite number $k$, then $n$ is also divisible by any prime factor $p$ of $k$. For such a $p$, we have $1 < p < k \leq \sqrt{n}$. Thus, we can restrict our divisibility testing to prime numbers $p$, $1 < p \leq \sqrt{n}$.

Example: we will prove that 47 is prime. Since $36 < 47 < 49$, $6 < \sqrt{47} < 7$, so we only have to test divide 47 by the prime numbers $p$, $1 < p \leq 6$. There are only three such prime numbers: 2,3,5.

47 is not divisible by 2 because it is even. 47 is not divisible by 3 because the sum of the digits of 47, 11, is not divisible by 3. 47 is not divisible by 5 because its last digit is neither 0 nor 5. Therefore, 47 is prime.

Observe that if we do not have a list of prime numbers available, we can also establish primality of $n$ by test dividing $n$ by all integers $k$ with $1 < k \leq \sqrt{n}$. This slightly cruder algorithm takes a little longer – it requires roughly $\sqrt{n}$ test divisions – but it has the advantage that it does not require a list of primes in the range 2 to $\sqrt{n}$ as input.
A crude primality test by trial division

This is the algorithm for primality testing discussed on the last page in pseudocode:

```plaintext
primality_test(int n) {
    int k = 2;
    int upper_limit = floor(sqrt(n));
    while (k <= upper_limit AND n mod k > 0)
        k++;
    if (k==upper_limit)
        return(n is prime);
    else
        return(n is not prime);
}
```
Prime Factorization

According to what we just learned, a composite number $n$ can be factored into a product of two positive integers that are both smaller than $n$. Then each of those numbers is either prime itself, or composite, in which case it can be factored again into a product of smaller integers.

This factorization process must end after finitely many steps with a **prime factorization** – a representation of $n$ as a product of prime numbers. It is a theorem, called the **fundamental theorem of arithmetic**, that each composite number $n$ has a prime factorization, and that this prime factorization is **unique except for rearrangement**, i.e. two different prime factorizations of $n$ must contain the same factors, only possibly in a different order.

An equivalent statement of the fundamental theorem of arithmetic is that every integer greater than 1 is either prime, or a product of primes, where the product is unique except for rearrangement.

Since the prime factorization may contain the same prime factor more than once, the general form of a prime factorization is $p_1^{k_1} p_2^{k_2} \ldots p_n^{k_n}$ where $p_1, \ldots, p_n$ are prime numbers and $k_1, \ldots, k_n$ are positive integers.
Example of Prime Factorization

The recursive factorization process we described on the previous slide can be visualized with a tree diagram like the one on the right.

\[
120 = 12 \cdot 10 \\
= (3 \cdot 4) \cdot (2 \cdot 5) \\
= (3 \cdot 2 \cdot 2) \cdot (2 \cdot 5) \\
= 2 \cdot 2 \cdot 2 \cdot 3 \cdot 5 \\
= 2^3 \cdot 3 \cdot 5.
\]

We could have used a different sequence of factorizations, such as \(120 = 2 \cdot 60 = 2 \cdot 2 \cdot 30 = 2 \cdot 2 \cdot 2 \cdot 15 = 2 \cdot 2 \cdot 2 \cdot 3 \cdot 5\) and arrived at the same prime factorization.
Prime Factorization is Computationally Expensive

There are no known “efficient” classical algorithms to find prime factorizations of large numbers with large prime factors. It takes a computer only fractions of a second to multiply two 500 digit primes to form a composite number with two prime factors that has 1000 digits. However, to factor the 1000 digit number and to recover the two primes would be impractical with current (2019) technology.
There are Infinitely Many Primes

It has been known since ancient times that there are infinitely many primes. We will discuss Euclid’s famous proof by contradiction here.

We will assume, to get a contradiction, that there are only finitely many primes. In that case, we can list all of them: \( p_1, p_2, ..., p_n \). We now consider the number

\[
P = p_1 p_2 ... p_n + 1.
\]

Each \( p_k \) is greater than 1, so \( P > p_k \) for each \( k = 1..n \). Since the \( p_k \) are all the prime numbers, and \( P \) is not equal to any of them, \( P \) is not prime. Thus \( P \) is composite. By the fundamental theorem of arithmetic, \( P \) must then be a product of primes. But \( P \) has a remainder of 1 with respect to division by each of the prime numbers \( p_k \), hence \( P \) is not divisible by any prime number. That is a contradiction.

Therefore, the assumption that there are only finitely many primes was false. This completes the proof that there are infinitely many primes.
The Distribution of Primes and the Prime Number Theorem

There is no known formula that outputs prime numbers on demand. In the sequence of integers, prime numbers seem to appear “randomly”. Yet at the same time, the total number of primes less than or equal an integer $n$ approximately follows a law that is given by the prime number theorem.

The prime number theorem, non-rigorously stated, says that the number of primes not exceeding $n$ is approximately $n/\ln n$.

Example 1: There are 25 primes that are less than or equal to $n = 100$, and $\frac{100}{\ln 100} \approx 21.7$.

Example 2: There are 20 primes $n$ that satisfy $174 \leq n \leq 281$. We can approximate this value using the prime number theorem as $\frac{281}{\ln 281} - \frac{173}{\ln 173} \approx 16.3$. 
The Greatest Common Divisor (I)

The greatest common divisor (gcd) of two positive integers \(a, b\), written \(\text{gcd}(a, b)\), is the largest integer that divides both \(a\) and \(b\). Examples:

\[
\text{gcd}(12, 18) = 6 \\
\text{gcd}(9, 8) = 1
\]

When \(\text{gcd}(a, b) = 1\), we call \(a\) and \(b\) relatively prime (to each other). We can find \(\text{gcd}(a, b)\) from the prime factorizations of \(a\) and \(b\). We will explain that by example first. Take

\[
a = 24 = 2 \cdot 2 \cdot 2 \cdot 3 \\
b = 18 = 2 \cdot 3 \cdot 3
\]

The gcd of \(a\) and \(b\) can only have as many of each prime factor as the two numbers have in common. Since \(a\) has three factors 2, but \(b\) only one, \(\text{gcd}(a, b)\) has only one factor 2. Since \(a\) has only one factor 3, and \(b\) has two, \(\text{gcd}(a, b)\) has only one factor 3. Therefore, \(\text{gcd}(a, b) = 2 \cdot 3 = 6\).

When \(a\) divides \(b\), then \(\text{gcd}(a, b) = a\).
The Greatest Common Divisor (II)

Let us generalize what we just learned. Suppose

\[ a = p_1^{k_1} \ldots p_n^{k_n} \]
\[ b = p_1^{m_1} \ldots p_n^{m_n} \]

where \( p_1, \ldots, p_n \) are prime numbers and the exponents \( k_1, \ldots, k_n, m_1, \ldots m_n \) are non-negative integers. We allow for some of the exponents to be zero to handle the case of prime factors that are exclusive to \( a \) or \( b \). Then \( \gcd(a, b) = p_1^{\min\{k_1,m_1\}} \ldots p_n^{\min\{k_n,m_n\}} \).

The notation \( \min\{x, y\} \) stands for the smallest of the two numbers \( x, y \). Note that when \( x = y \), \( \min\{x, y\} = x = y \).
The Least Common Multiple (1)

The least common multiple (lcm) of two positive integers \( a, b \), written \( \text{lcm}(a, b) \), is the smallest integer that is a multiple of both \( a \) and \( b \). Examples:

\[
\text{lcm}(6,4) = 12 \\
\text{lcm}(2,3) = 6
\]

We can find \( \text{lcm}(a, b) \) from the prime factorizations of \( a \) and \( b \). Let us take an example first:

\[
a = 12 = 2 \cdot 2 \cdot 3 = 2^2 \cdot 3 \\
b = 18 = 2 \cdot 3 \cdot 3 = 2 \cdot 3^2
\]

Then any common multiple of \( a \) and \( b \) must have at least two powers of 2, and at least two powers of 3. Since \( 2^23^2 \) is a common multiple of \( a \) and \( b \), it must be the least common one. Therefore, \( \text{lcm} (a, b) = 2^23^2 = 36 \). This illustrates the rule that we find the lcm of two positive integers from their prime factorization by selecting the \textbf{maximum} of the two exponents for each prime factor.
The Least Common Multiple (2)

Let us state this rule more formally. Suppose again like on the previous slide that the prime factorizations of $a$ and $b$ are

$$a = p_1^{k_1} \ldots p_n^{k_n}$$
$$b = p_1^{m_1} \ldots p_n^{m_n}.$$

Then any multiple of $a$ and $b$ must have at least as much of each prime factor as $a$ and $b$ individually. Therefore,

$$\text{lcm}(a, b) = p_1^{\max\{k_1, m_1\}} \ldots p_n^{\max\{k_n, m_n\}}$$

The notation $\max\{x, y\}$ stands for the largest of the two numbers $x, y$. Note that when $x = y$, $\max\{x, y\} = x = y$. 
The Relationship between Gcd and Lcm

We have shown that if

\[ a = p_1^{k_1} \ldots p_n^{k_n} \]
\[ b = p_1^{m_1} \ldots p_n^{m_n} \]

then

\[ \text{gcd}(a, b) = p_1^{\min\{k_1, m_1\}} \ldots p_n^{\min\{k_n, m_n\}} \]
\[ \text{lcm}(a, b) = p_1^{\max\{k_1, m_1\}} \ldots p_n^{\max\{k_n, m_n\}} \]

When we multiply the two equations, we get

\[ \text{gcd}(a, b) \text{lcm}(a, b) = p_1^{\min\{k_1, m_1\} + \max\{k_1, m_1\}} \ldots p_n^{\min\{k_n, m_n\} + \max\{k_n, m_n\}} \]

For any two numbers \( a \) and \( b \), \( \min\{a, b\} + \max\{a, b\} = a + b \). Therefore,

\[ \text{gcd}(a, b) \text{lcm}(a, b) = p_1^{k_1+m_1} \ldots p_n^{k_n+m_n} = ab. \]

We have just proved that \( \text{gcd}(a, b) \text{lcm}(a, b) = ab \). One consequence of this equation is that if \( a \) and \( b \) are relatively prime to each other, then \( \text{lcm}(a, b) = ab \).
The Euclidean Algorithm (I)

For large numbers $a, b$, it may be computationally prohibitive to find the prime factorizations. Fortunately, there is an efficient way to compute $\gcd(a, b)$ that does not rely on the prime factorizations, called the Euclidean Algorithm.

To understand the algorithm, we will have to recall a fact from a previous presentation: If $a, b$ and $c$ are integers, then, if $b$ and $c$ are both multiples of $a$, so is $b + c$, and if $b$ is a multiple of $a$, so is any multiple of $b$.

The Euclidean Algorithm is based on the following theorem: If $a, b, q, r$ are integers and $a = bq + r$, then $\gcd(a, b) = \gcd(b, r)$.

Proof: suppose $c$ is a common divisor of $b$ and $r$. Then $c$ also divides $bq$, and therefore divides $a = bq + r$. Thus $c$ is a common divisor of $a$ and $b$.

Now suppose $c$ is a common divisor of $a$ and $b$. Then $c$ also divides $-bq$, and therefore also divides $r = a - bq$. Thus $c$ is a common divisor of $b$ and $r$.

We have proved that the set of common divisors of $a$ and $b$ and the set of common divisors of $b$ and $r$ are equal. Therefore, the greatest common divisor of $a$ and $b$ and the greatest common divisor of $b$ and $r$ must be equal as well.
The Euclidean Algorithm (II)

The Euclidean Algorithm to find \( \gcd(a, b) \) consists of repeated application of the division algorithm \( a = bq + r \), where after each step, \( b \) becomes the new \( a \), and \( r \) becomes the new \( b \).

Example: the Euclidean Algorithm with \( a = 102 \) and \( b = 30 \).

\[
102 = 3 \cdot 30 + 12
\]

After one step, we know that \( \gcd(102, 30) = \gcd(30, 12) \).

\[
30 = 2 \cdot 12 + 6
\]

Now we know that \( \gcd(102, 30) = \gcd(30, 12) = \gcd(12, 6) \).

At this point, we can stop because 12 is a multiple of 6, so we know \( \gcd(12, 6) = 6 \). Since 12 is a multiple of 6, the remainder in the next division step is zero:

\[
12 = 2 \cdot 6 + 0
\]

This teaches us the halting condition of the Euclidean Algorithm: when we reach a remainder of zero, the previous (non-zero) remainder is the gcd.
The Euclidean Algorithm (III)

Let us now give a general description of the Euclidean Algorithm and explain why it always terminates after finitely many steps.

Let us set $a = r_0$ and $b = r_1$. The $n$-th step (for $n \geq 1$) of the algorithm consists of application of the division algorithm

$$r_{n-1} = r_n q_n + r_{n+1}.$$ 

Each remainder $r_{n+1}$ satisfies $0 \leq r_{n+1} < r_n$ because $r_n$ acts as the divisor of the division algorithm. Since the remainders are also integers, $r_{n+1} < r_n$ implies that the remainders decrease by at least 1 with each step: $r_{n+1} \leq r_n - 1$. On the other hand, all remainders are non-negative. Hence, in finitely many steps, one remainder must reach the value zero: $r_{n+1} = 0$. For that $n$, $r_{n-1} = r_n q_n$. Thus

$$\gcd(a, b) = \gcd(r_{n-1}, r_n) = r_n.$$ 

That confirms that the last non-zero remainder is the gcd.
The Euclidean Algorithm (IV)

An interesting phenomenon happens when we apply the Euclidean Algorithm to two consecutive terms of the Fibonacci sequence. Let us take $a = 21$ and $b = 13$ as an example:

\[
\begin{align*}
21 &= 1 \cdot 13 + 8 \\
13 &= 1 \cdot 8 + 5 \\
8 &= 1 \cdot 5 + 3 \\
5 &= 1 \cdot 3 + 2 \\
3 &= 1 \cdot 2 + 1 \\
2 &= 2 \cdot 1 + 0
\end{align*}
\]

Observe that the remainders reproduce the lower terms of the Fibonacci sequence as we defined it in the lecture on sequences, starting at $f_1$, and that all quotients are 1, except the last one, which is 2.
Performance of the Euclidean Algorithm (I)

Let us compare the calculation we just made to ones with the same $a = 21$, but different $b$ values, from $b = 1$ to $b = 21$.

<table>
<thead>
<tr>
<th>$b = 1$</th>
<th>$b = 2$</th>
<th>$b = 3$</th>
<th>$b = 4$</th>
<th>$b = 5$</th>
<th>$b = 6$</th>
<th>$b = 7$</th>
<th>$b = 8$</th>
<th>$b = 9$</th>
<th>$b = 10$</th>
<th>$b = 11$</th>
<th>$b = 12$</th>
<th>$b = 13$</th>
<th>$b = 14$</th>
<th>$b = 15$</th>
<th>$b = 16$</th>
<th>$b = 17$</th>
<th>$b = 18$</th>
<th>$b = 19$</th>
<th>$b = 20$</th>
<th>$b = 21$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$21 = 1 \cdot 15 + 6$</td>
<td>$21 = 1 \cdot 16 + 5$</td>
<td>$21 = 1 \cdot 17 + 4$</td>
<td>$21 = 1 \cdot 18 + 3$</td>
<td>$21 = 1 \cdot 19 + 2$</td>
<td>$21 = 1 \cdot 20 + 1$</td>
<td>$21 = 1 \cdot 21 + 0$</td>
<td>$21 = 2 \cdot 8 + 5$</td>
<td>$21 = 2 \cdot 9 + 3$</td>
<td>$21 = 2 \cdot 10 + 1$</td>
<td>$21 = 2 \cdot 11 + 10$</td>
<td>$21 = 2 \cdot 12 + 9$</td>
<td>$21 = 2 \cdot 13 + 8$</td>
<td>$21 = 2 \cdot 14 + 7$</td>
<td>$21 = 2 \cdot 7 + 1$</td>
<td>$21 = 3 \cdot 3 + 0$</td>
<td>$21 = 4 \cdot 5 + 1$</td>
<td>$21 = 3 \cdot 6 + 1$</td>
<td>$21 = 3 \cdot 7 + 0$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notice that the algorithm performs exceptionally poorly in the Fibonacci case, where it needs 6 steps. In all other cases, except $b = 8$, it terminates in 3 steps or less. The case $b = 8$ also goes through Fibonacci numbers as the remainders. These examples suggest that the Fibonacci numbers represent a worst-case scenario for the performance of the Euclidean algorithm.
Performance of the Euclidean Algorithm (II)

Let us consider the general case of the Euclidean Algorithm, applied to integers $a$ and $b$ with $0 < b < a$. This means that all quotients will be positive numbers.

Let us look at the algorithm in reverse order, and consider the last step first, where the remainder is zero. If there are $n$ steps total, then the last step is $r_{n-1} = r_n q_n + 0$.

Generally, quotients can be 1, but the last step is an exception, because $q_n = 1$ would imply $r_{n-1} = r_n$, violating the condition $r_n < r_{n-1}$. Therefore, $q_n \geq 2$, and thus $r_{n-1} \geq 2r_n$. We also know that $r_n$ cannot be zero, because $r_{n+1} = 0$, so $r_n \geq 1$ and thus $r_{n-1} \geq 2$. Observe that since $f_1 = 1$ and $f_2 = 2$, we just showed $r_n \geq f_1$ and $r_{n-1} \geq f_2$.

Now consider the second to last step: $r_{n-2} = r_{n-1} q_{n-1} + r_n$. We know $q_{n-1} \geq 1$, so $r_{n-2} \geq r_{n-1} + r_n \geq 2 + 1 = 3 = f_3$. Note that $r_{n-2} \geq r_{n-1} + r_n$ is an inequality version of the recurrence relation for the Fibonacci sequence.

We could turn this argument into a formal inductive proof that $r_{n-k} \geq f_{k+1}$ for all $k = 0, \ldots, n$. What this is saying is that the remainders (taken in reverse order) grow at least as fast as the Fibonacci sequence.
The cases $k = n$ and $k = n - 1$ (i.e. the inequalities for the first two steps) are of particular interest: $r_0 \geq f_{n+1}$ and $r_1 \geq f_n$. Keeping in mind that $a = r_0$ and $b = r_1$, we get $a \geq f_{n+1}$ and $b \geq f_n$. In the lecture on structural induction, we will prove that $f_n \geq \left(\frac{3}{2}\right)^{n-1}$ for all $n \geq 0$. Combining this with the inequalities above, we get $a \geq \left(\frac{3}{2}\right)^n$ and $b \geq \left(\frac{3}{2}\right)^{n-1}$. Using the logarithm, we can solve each inequality for $n$ and get $n \leq \frac{\log_{10} a}{\log_{10} \frac{3}{2}} = A$ and $n \leq \frac{\log_{10} b}{\log_{10} \frac{3}{2}} + 1 = B$.

Here we introduced $A$ and $B$ as convenient names for the two upper bounds on $n$. Observe that since $n$ is an integer, the two inequalities imply something slightly stronger: $n \leq \lfloor A \rfloor$ and $n \leq \lfloor B \rfloor$.

For example, if $a = 21$, then $A = 7.5087...$. Therefore, the Euclidean algorithm will take at most 7 steps, no matter what $b$ is ($0 < b < a$).
Performance of the Euclidean Algorithm (IV)

With a little algebra, we can work out when \([B]\) provides a better estimate for \(n\):

- If \(\frac{2}{3}a \leq b \leq a\), then \([B] = [A]\) or \([B] = [A] + 1\). In other words, in this case, \(n \leq [A]\) is the best we can do, but \([B]\) is at most one more.

- If \(\frac{4}{9}a < b < \frac{2}{3}a\), then \(A - 1 < B < A\), which implies \([B] = [A]\) or \([B] = [A] - 1\). In this case, \(n \leq [B]\) is the best we can do, though \([B]\) may still be equal to \([A]\). This happens when \(b\) is fairly close to \(\frac{2}{3}a\).

- Thus, for \(\frac{4}{9}a < b \leq a\), the two estimates differ at most by one, and are thus essentially in agreement if \(a\) is large.

- If \(b \leq \frac{4}{9}a\), then \(B \leq A - 1\). This will guarantee \([B] < [A]\), i.e. \(n \leq [B]\) is the better estimate, though this could already have occurred at some higher \(b\) value.

The table on the right side illustrates all this with \(a = 21\) and \(2 \leq b \leq 20\). Recall that \([A] = 7\).

For \(b\) values just a little below \(a\), \([B]\) gives us the estimate \(n \leq 8\) instead of \(n \leq 7\).

For \(b\) in the vicinity of \(\frac{2}{3}a = 14\), the two estimates are agree.

At some \(b\) value between 14 and \(\frac{4}{9}a = 9.33\), the \([B]\) estimate becomes the better one, here at \(b = 11\).
Performance of the Euclidean Algorithm (V)

We can improve our estimate a little if we use a better inequality for the Fibonacci sequence. We used the estimate $f_n \geq \left(\frac{3}{2}\right)^{n-1}$ because it simplified the subsequent discussion. However, as is shown in the expanded notes on Recursion and Structural Induction, a slightly better estimate holds:

\[ f_n \geq \left(\frac{1 + \sqrt{5}}{2}\right)^{n-1} \quad \text{for all } n \geq 0. \]

It is traditional to use the letter $\varphi = \frac{1 + \sqrt{5}}{2}$ here.

The entire argument we just made works the same way now, with $\varphi$ playing the role of the number $3/2$. We obtain $n \leq \log_{10} a / \log_{10} \varphi = A_\varphi$ and $n \leq \log_{10} b / \log_{10} \varphi + 1 = B_\varphi$.

In our example $a = 21$, $A_\varphi = 6.32\ldots$, so we get the improved estimate that the algorithm will take at most 6 steps. Since it actually takes 6 steps for $n = 13$, we cannot get a better estimate that is independent of $b$. For the improved $B_\varphi$ values, see the table on the right. Observe that $\lfloor B_\varphi \rfloor = n$ in the two special “worst” cases we previously identified (where remainders reproduce parts of the Fibonacci sequence): $b = 13$ and $b = 8$. 

| $b$ | $n$ | $|B|$ | $|B_\varphi|$ |
|-----|-----|-----|-------------|
| 20  | 2   | 8   | 7           |
| 19  | 3   | 8   | 7           |
| 18  | 2   | 8   | 7           |
| 17  | 3   | 7   | 6           |
| 16  | 3   | 7   | 6           |
| 15  | 3   | 7   | 6           |
| 14  | 2   | 7   | 6           |
| 13  | 6   | 7   | 6           |
| 12  | 3   | 7   | 6           |
| 11  | 3   | 6   | 5           |
| 10  | 2   | 6   | 5           |
| 9   | 2   | 6   | 5           |
| 8   | 5   | 6   | 5           |
| 7   | 1   | 5   | 5           |
| 6   | 2   | 5   | 4           |
| 5   | 2   | 4   | 4           |
| 4   | 2   | 4   | 3           |
| 3   | 1   | 3   | 3           |
| 2   | 2   | 2   | 2           |
Performance of the Euclidean Algorithm (VI)

We can now derive a convenient rule of thumb from the previous discussion. Since \( B_\phi \) is usually smaller than \( A_\phi \), and at worst one bigger (still assuming that \( a \) and \( b \) are integers with \( 0 < b < a \)), we can just use the estimate \( n \leq B_\phi \) to get

\[
 n \leq \frac{\log_{10} b}{\log_{10} \phi} + 1 < 4.785 \log_{10} b + 1.
\]

We know from *Integer Representations* that the number \( k \) of decimal digits of \( b \) is \( k = \lceil \log_{10} b \rceil + 1 \).

We also know from *Functions* that \( x - 1 < \lfloor x \rfloor \) for any real number \( x \). Thus \( k - 1 = \lfloor \log_{10} b \rfloor > \log_{10} b - 1 \). Hence \( \log_{10} b < k \) and

\[
 n < 4.785 \log_{10} b + 1 < 4.785k + 1.
\]
Performance of the Euclidean Algorithm (VII)

This means that the number of steps in the Euclidean algorithm, applied to positive integers $a$ and $b$ ($a < b$), is less than 5 times the number of decimal digits of $b$ (and actually less than 4.785 times that number.)

This is very good news for the performance of the Euclidean algorithm. Due to the logarithm in the estimates for $n$, the number of steps required is reasonably small even for large numbers $a$, $b$. For example, if $a$, $b$ both have 1000 decimal digits, the Euclidean algorithm will require less than 4785 steps. The calculation would finish essentially instantly even on a modest contemporary computer.

By contrast, to find the gcd of two 1000 decimal digit numbers via prime factorization could take an unreasonably long time, or be entirely impractical if all the prime factors are “large” (think all prime factors of each number have several hundred digits.)
The Euler Phi Function (1)

Recall that two integers $a, b$ are called *relatively prime* or *coprime* (to each other) iff $\gcd(a, b) = 1$. Equivalently, two integers are relatively prime to each other iff they have no common prime factor.

For example, 25 and 27 are relatively prime to each other because 25 only has the prime factor 5, and 27 only has the prime factor 3.

The **Euler Phi Function** counts how many positive integers are relatively prime to a given positive integer. More precisely, $\varphi(n)$ is the number of integers from 1 to $n$ that are relatively prime to $n$.

For example, $\varphi(12) = 4$ because 1, 5, 7 and 11 are the integers in the range from 1 to 12 that are relatively prime to 12.

Observe that since 1 is always relatively prime to $n$, $\varphi(n)$ can be at most $n - 1$. This is the case when $n$ is prime.
The Euler Phi Function (2)

The Euler Phi Function is connected to a theorem that is important in number theory and in cryptography, known as Euler’s Theorem:

If \( a \) and \( m \) are relatively prime, then \( a^{\phi(m)} \equiv 1 \mod m \).

This means that when we are doing modular arithmetic in \( \mathbb{Z}_m \), any number \( a \in \mathbb{Z}_m \) that is relatively prime to \( m \) is a root of unity, i.e raising \( a \) to some positive integer power will produce the number 1. Euler’s theorem predicts that \( \phi(m) \) is one such power.

Let us take \( m = 10 \) as an example. \( \phi(10) = 4 \) because there are exactly 4 numbers from 1 to 9 that are relatively prime to 10. They are 1, 3, 7 and 9.

The number \( a = 3 \) is relatively prime to 10. Therefore, Euler’s theorem correctly predicts that \( a^{\phi(m)} \mod m = 3^4 \mod 10 = 1 \).

This seeming mathematical curiosity has important consequences.
The Euler Phi Function (3)

Euler’s theorem tells us that in $\mathbb{Z}_m$, numbers $a$ that are relatively prime to $m$ have a multiplicative inverse and even supplies a formula for it.

For example, we saw that $3^4 \mod 10 = 1$. This means that $3^3 \equiv 7$ is the multiplicative inverse of 3 in $\mathbb{Z}_{10}$, because $3^3 \cdot 3^1 = 3^4$, and $3^4 \equiv 1 \mod 10$.

We can verify this directly: $3 \cdot 7 = 21 \equiv 1 \mod 10$.

Generally, if $a$ is relatively prime to $m$, its inverse in $\mathbb{Z}_m$ is $a^{\phi(m)-1} \mod m$.

In the extended lecture on modular arithmetic, we proved that if $a \in \mathbb{Z}_m$ is invertible, then $\gcd(a, m) = 1$. We were not able to prove the converse then. Now we see that the converse is a direct consequence of Euler’s theorem: for all $a \in \mathbb{Z}_m$, if $\gcd(a, m) = 1$, then $a$ is invertible.

Let us now discuss what this means in the special case when $m$ is prime.
The Euler Phi Function (4)

When \( m \) is prime, then every number from 1 to \( m - 1 \) is relatively prime to \( m \), so then every nonzero number \( a \in \mathbb{Z}_m \) has a multiplicative inverse. In fact, based on what we just learned, and because \( \phi(m) = m - 1 \) in this case, the inverse is

\[
a^{-1} = a^{m-2} \mod m.
\]

This is a big deal. **It means that when \( m \) is prime, we can do division in \( \mathbb{Z}_m \) just like in \( \mathbb{Q} \) and \( \mathbb{R} \), i.e. we can divide by any nonzero number.** Since we were able to do addition, subtraction and multiplication anyway, we can do the four arithmetic operations just like in \( \mathbb{Q} \) and \( \mathbb{R} \).

In abstract algebra, we call a set of numbers where we can do the four arithmetic operations like in \( \mathbb{Q} \) and \( \mathbb{R} \) a **field**.

Rephrased using this term, if \( m \) is prime, then \( \mathbb{Z}_m \) is a field.

On the next page, we pick the prime \( m = 29 \) and solve a division problem to illustrate.
The Euler Phi Function (5)

We just said that in $\mathbb{Z}_{29}$, we can divide by any nonzero element. Let us find $b = 27/23$ in $\mathbb{Z}_{29}$. What we really mean by that is, let us find a number $x \in \mathbb{Z}_{29}$ so that

$$23x = 27.$$

Since arithmetic in $\mathbb{Z}_{29}$ works just like real number division, we find $x$ by multiplying both sides by the multiplicative inverse of 23.

The Euler phi function will do most of the heavy lifting for us to find that inverse. Since $m$ is prime, $\phi(29) = 28$. By Euler’s theorem, $23^{28} \mod 29 = 1$. Therefore, $23^{27} \mod 29 = 24$ is the multiplicative inverse of 23. (Thanks to fast modular exponentiation, finding this 24 is not too bad computationally.)

Therefore, $x = 24 \cdot 27 \mod 29 = 10$. We have found that $\frac{27}{23} = 10$ in $\mathbb{Z}_{29}$.

We can verify this directly through multiplication: $10 \cdot 23 = 230 = 7 \cdot 29 + 27$, hence $10 \cdot 23 \equiv 27 \mod 29$. 

There is no obvious shortcut formula for finding multiplicative inverses in \( \mathbb{Z}_m \), aka modular multiplicative inverses. They seem to be “random”, other than having the symmetry \( b = a^{-1} \leftrightarrow a = b^{-1} \).

On the right is a table of multiplicative inverses in \( \mathbb{Z}_{11}^* \) to illustrate. (The raised star indicates that we omit the number zero from the set. Zero has no multiplicative inverse.)

Euler’s theorem lets us compute modular multiplicative inverses by a formula. However, the exponents involved can be almost as high as \( m \) itself. As we have seen, the inverse of \( a \in \mathbb{Z}_m \), when \( m \) is prime, is \( a^{m-2} \mod m \).

With fast modular exponentiation, the cost of performing modular exponentiation is not too bad, even for large exponents, but fortunately, there is an even faster method available for solving the modular multiplicative inverse problem. That method is based on an extension of the Euclidean Algorithm, aptly called the Extended Euclidean Algorithm.
The Extended Euclidean Algorithm (1)

Consider the following example of the Euclidean Algorithm with \( a = 30 \) and \( b = 7 \):

\[
30 = 4 \cdot 7 + 2 \\
7 = 3 \cdot 2 + 1.
\]

We can stop when the remainder is 1 because at that point, we know that \( \gcd(30,7) = 1 \).

Now let us isolate the remainders in each of the equations, starting with the last equation:

\[
1 = 7 - 3 \cdot 2 \\
2 = 30 - 4 \cdot 7.
\]

We can substitute the equation for the remainder 2 into the first equation and simplify:

\[
1 = 7 - 3 \cdot (30 - 4 \cdot 7) = 13 \cdot 7 + (-3) \cdot 30.
\]

We have thus expressed \( \gcd(30,7) \) as an integer multiple of 30, plus an integer multiple of 7.
The Extended Euclidean Algorithm (2)

What we just did is called the *Extended Euclidean Algorithm*. We carried out the Euclidean algorithm until we got the gcd as the remainder, and then we **solved each equation for the remainder**. Then we worked backwards, substituting the equation for each remainder into the previous equation and simplifying. When we were back at the top, we had the equation $\gcd(a, b) = ax + by$ for some coefficients $x, y$.

The numbers $x, y$ are also known as the *Bézout coefficients*.

The fact that we can always carry out the extended Euclidean algorithm proves **Bézout’s Theorem**, which says that for any positive integers $a, b$, we can always find integers $x, y$ so that

$$\gcd(a, b) = ax + by.$$
The Extended Euclidean Algorithm (3)

An important special case of Bézout’s Theorem is that if the positive integers $a, b$ are also relatively prime to each other, we can always find integers $x, y$ so that

$$1 = ax + by.$$ 

This allows us to solve the modular multiplicative inverse problem very efficiently. Suppose we have a positive $a \in \mathbb{Z}_m$ that is relatively prime to $m$. By Bézout’s Theorem, we can find integers $x, y$ so that $1 = ax + my$. We apply mod $m$ to the equation. This turns $my$ into zero and yields:

$$a(x \mod m) \mod m = 1.$$ 

This means $a^{-1} = x \mod m$.

We just accomplished two things. We proved that $a \in \mathbb{Z}_m$ is invertible if it is relatively prime to $m$, without relying on the “black box” of Euler’s theorem, which we have not proved. Additionally, we produced a simple and computationally efficient method for finding the multiplicative inverse that does not require values of the Euler phi function.
Let us illustrate this method by finding the multiplicative inverse of 35 in $a \in \mathbb{Z}_{64}$. There is no need to confirm ahead of time that 35 and 64 are relatively prime to each other. This confirmation will come for free out of the extended Euclidean Algorithm:

$$
\begin{align*}
64 &= 1 \cdot 35 + 29 \\
35 &= 1 \cdot 29 + 6 \\
29 &= 4 \cdot 6 + 5 \\
6 &= 1 \cdot 5 + 1
\end{align*}
$$

Back-substituting, we get:

\begin{align*}
1 &= 6 - 1 \cdot (29 - 4 \cdot 6) = 5 \cdot 6 - 1 \cdot 29 \\
1 &= 5 \cdot (35 - 1 \cdot 29) - 1 \cdot 29 = 5 \cdot 35 - 6 \cdot 29 \\
1 &= 5 \cdot 35 - 6 \cdot (64 - 1 \cdot 35) = 11 \cdot 35 - 6 \cdot 64.
\end{align*}

We conclude that $35^{-1} = 11$ in $\mathbb{Z}_{64}$.

Knowing $35^{-1}$ in $\mathbb{Z}_{64}$, we now also have a formulaic solution for $35x = c$, for any $c \in \mathbb{Z}_{64}$: $x = 11 \cdot c \mod 64$. 

The Extended Euclidean Algorithm (4)
The Extended Euclidean Algorithm (5)

We could also have found the multiplicative inverse of 35 in $\mathbb{Z}_{64}$ by using Euler’s theorem. For that, we would have to find $\phi(64)$. Finding $\phi$ of a number of the form $2^n$ is easy, because the positive numbers relatively prime to $2^n$ are precisely the odd numbers $1, 3, 5, \ldots, 2^n - 1$, which is every second number in the range from 1 to $2^n$.

Therefore, $\phi(64) = 32$.

This means that $35^{-1}$ in $\mathbb{Z}_{64}$ is $35^{31} \mod 64 = 11$. 