We’re now going to talk about Turing machines (TM). So recall the Turing machine model.

The default model uses a single, one-sided tape, potentially unbounded to the right, divided into cells. Each cell can hold one symbol from the tape alphabet Γ.

The TM has a finite control. At each step, depending on the current state, the TM can rewrite the current cell, and move left (L), right (R), or not move at all (S).

If it moves off the left edge, the TM “crashes” and has no next move (and does not accept its input).

There is a single accept (or “halting”) state $h$. 

Turing machines
All cells except the input initially contain the distinguished symbol B (for “blank”).

The initial configuration is

\[ B a_1 a_2 \cdots a_t BBB \cdots , \]

where \( x = a_1 a_2 \cdots a_t \) is the input.

The TM accepts its input if, when started in the initial configuration, scanning cell 0 with a blank, it eventually reaches \( h \). Otherwise it does not accept.

It does not have to read all its input to accept it.
More about Turing machines

This is the default model.

As you have seen in CS 360, 365, or a similar course elsewhere, most variations on the basic model have exactly the same computing power:

- two-sided tape
- multiple tapes
- nondeterminism

And remember the universal TM: it takes the encoding of a TM $T$ and a word $x$ as an input and simulates the action of $T$ on $x$. 
A basic philosophical and mathematical question: when is a string of symbols “complex”?

Intuitively, a string such as

0101010101010101010101010

is not complex, because there is an easy way to describe it, whereas a string such as

0110101011001110010001010

is complex because it appears to have no simple description.
Here’s another way to think about it.

Suppose I flip a fair coin, recording 0 for heads and 1 for tails. If I produce

010101011001110010001010,  

and claim it is a record of 25 tosses, no one would be very surprised.

But a few eyebrows would be raised if I produced

0101010101010101010101010

as my record of tosses. But why? Both outcomes have equal probability!

This “paradox” shows that the notions of complexity and randomness are linked.
James Boswell (1740–1795), the biographer of lexicographer and essayist Samuel Johnson, wrote this about the events of June 24, 1784:

*I recollect nothing that passed this day, except Johnson’s quickness, who, when Dr. Beattie observed, as something remarkable which had happened to him, that he had chanced to see both No. 1 and No. 1000, of the hackney-coaches, the first and the last; ‘Why, Sir, (said Johnson,) there is an equal chance for one’s seeing those two numbers as any other two.’ He was clearly right; yet the seeing of the two extremes, each of which is in some degree more conspicuous than the rest, could not but strike one in a stronger manner than the sight of any other two numbers.*
In 1819, Laplace wrote

This is the place to define the word extraordinary. We arrange in our thought all possible events in various classes; and we regard as extraordinary those classes which include a very small number. Thus at the play of heads and tails the occurrence of heads a hundred successive times appears to us extraordinary because of the almost infinite number of combinations which may occur in a hundred throws; and if we divide the combinations into regular series containing an order easy to comprehend, and into irregular series, the latter are incomparably more numerous.
Kolmogorov complexity is a way to measure the complexity, or randomness, of a finite string.

*Roughly speaking*, the Kolmogorov complexity $C(x)$ of a string $x$ is the size (number of bits) in the shortest Java program $P + \text{input } i$ that will print $x$ and then halt. (If you don’t like Java, feel free to substitute C, Python, or your favorite programming language.)

If the Kolmogorov complexity of a string $x$ is small, then there is a simple way to describe $x$. If the Kolmogorov complexity of $x$ is large, then $x$ is hard to describe; we say it is “complex”, “random”, or possesses “high information content”.
### Synonyms

<table>
<thead>
<tr>
<th>High Kolmogorov Complexity</th>
<th>Low Kolmogorov Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>complicated</td>
<td>clear</td>
</tr>
<tr>
<td>random</td>
<td>ordered</td>
</tr>
<tr>
<td>high information</td>
<td>low information</td>
</tr>
<tr>
<td>complex</td>
<td>simple</td>
</tr>
<tr>
<td>irregular</td>
<td>regular</td>
</tr>
<tr>
<td>incompressible</td>
<td>compressible</td>
</tr>
</tbody>
</table>

All these words refer to the *description* of \( x \), not the amount of time it takes to verify or compute it.
Are the prime numbers complex?

We often think of the prime numbers as being complex, and certainly some of their properties are.

But for our purposes, their Kolmogorov complexity is low in the following sense: consider a binary string of length $n$ with a 1 in position $i$ if $i$ is a prime, and 0 otherwise:

$$0110101000101\cdots.$$ 

What is the Kolmogorov complexity of this string?

Answer: it is $\leq \log_2 n + O(1)$.

Why? We can easily write a program that, given $n$ (which we can specify with $\log n + O(1)$ bits), outputs this string of length $n$.

So the primes are not complex in our sense.
Kolmogorov complexity was invented by Ray Solomonoff, who described it in papers in 1960 and 1964.

It was independently discovered by the Russian mathematician Andrey Nikolaevich Kolmogorov in 1965.

And it was also independently discovered by Gregory Chaitin (at the age of 18!) in 1966.

We can also view the combination of program and input \((P, i)\) as an optimal way to compress \(x\).

In this interpretation, instead of storing \(x\), we could store \((P, i)\), since we could always recover \(x\) by running \(P\) on input \(i\).

(Notice that this approach disregards the running time of \(P\) on input \(i\).)
Universal TM

For a more formal definition of Kolmogorov complexity, we need a universal Turing machine $U$.

The input to $U$ is a self-delimiting binary encoding of a Turing machine $T$, followed by $y \in \{0, 1\}^*$, the input for $T$.

“Self-delimiting” means that given $e(T)y$ we can tell where the encoding of $T$ ends and $y$ begins.

We assume that $T$’s input alphabet, as well as $U$’s, is $\{0, 1\}$. $U$ then simulates $T$ on input $y$. It is assumed that $T$ has an output tape, and the output of $U$ is what $T$ outputs if and when it halts.

Then $C(x)$ is formally defined to be the length of a shortest input $e(T)y$ that causes $U$ to output $x$. 
Theorem. We have $C(x) \leq |x| + O(1)$.

Note that the constant in the big-$O$ is independent of $x$.

Proof. Informally, we can use the following program:

```plaintext
program print(input);
    begin
        write(input);
    end.
```

Clearly the length of this program is $|x| + c$, where $c$ is the number of characters in the template above.

Formally, there exists some Turing machine $T$ that simply copies the input to the output. Then the input to $U$ is $e(T)x$, which is of length $|x| + |e(T)| = |x| + O(1)$. 

Kolmogorov complexity is not too large
Example.

Let’s show $C(xx) \leq C(x) + O(1)$.

Informally, given a program $P$ to print $x$, we simply call it twice to print $xx$.

The extra cost to build the “wrapper” program and call $P$ twice corresponds to the $O(1)$ term.

Can you prove $C(x) = C(xx) + O(1)$?
The invariance theorem

The next theorem, called the *invariance theorem*, shows that the particular choice of programming language or universal Turing machine is *irrelevant*, at least up to an additive constant.

**Theorem.** Suppose we define $C_{\text{Python}}$, $C_{\text{Java}}$, etc., analogously. Then we have, for example,

$$C_{\text{Java}}(x) \leq C(x) + O(1)$$

$$C(x) \leq C_{\text{Java}}(x) + O(1)$$

Thus all these measures are the same, up to an additive constant.
Proof of the invariance theorem

Proof. We prove $C(x) \leq C_{Java}(x) + O(1)$.

Suppose $C_{Java}(x) = d$.

Then there exists an Java program $P$ to print out $x$, of size $d$.

Now create a TM $T$ that is a Java interpreter; such a machine can be fed with $P$ to output $x$.

Thus

$$C(x) \leq |e(T)P| = |P| + |e(T)| = C_{Java}(x) + O(1).$$

To prove $C_{Java}(x) \leq C(x) + O(1)$, create a Turing machine simulator in Java and use the same argument.
We can think of the representation of a string $x$ as $e(T)y$ as a sort of optimal “compression” method (like the Unix compress command).

The $e(T)$ captures the “regular” aspects of $x$, while the $y$ captures the “irregular” aspects of $x$.

We call a string $x$ incompressible or random if $C(x) \geq |x|$. We cannot explicitly exhibit long incompressible strings $x$, but we can prove they exist:

**Theorem.** For all $n \geq 0$, there exists at least one string $x$ of length $n$ such that $C(x) \geq |x|$.

**Proof.** There are $2^n$ strings of length $n$, but at most $1 + 2 + \cdots + 2^{n-1} = 2^n - 1$ shorter descriptions.
More generally we have

**Proposition.** Let $S$ be a set of binary strings of cardinality $n$. Then some string $x \in S$ has $C(x) \geq \log_2(n + 1) - 1$.

**Proof.** Suppose $t$ is the largest Kolmogorov complexity of a string in $S$.

Then all strings in $S$ have Kolmogorov complexity $\leq t$.

Then there can be at most $1 + 2 + \cdots + 2^t = 2^{t+1} - 1$ different descriptions of strings in $S$.

So $n \leq 2^{t+1} - 1$.

In other words, $2^{t+1} \geq n + 1$, or $t + 1 \geq \log_2(n + 1)$, or $t \geq \log_2(n + 1) - 1$. 
Kolmogorov complexity is uncomputable

Unfortunately, Kolmogorov complexity is uncomputable, so perfect compression is unattainable.

**Theorem.** The quantity $C(x)$ is uncomputable.

**Proof.** Assume $C(x)$ is computable by a Turing machine $T$ that takes $x$ as input.

Create a new Turing machine $T'$ that (using $T$ as a subroutine) on input $\ell$, examines all strings of size $\ell$ in lexicographic order until it finds a string $y$ with $C(y) \geq |y| = \ell$.

Such a string exists by our theorem above. Then $T'$ outputs $y$. 
Now let’s compute the Kolmogorov complexity of $y$.

On the one hand, we have $C(y) \geq \ell$.

On the other hand, the string $y$ is completely determined by $T'$ and $\ell$, so $C(y) \leq |e(T')| + (\log_2 \ell) + 1$. Now

$$\ell \leq C(y) \leq |e(T)| + (\log_2 \ell) + c \quad (1)$$

for a constant $c$.

Choose $\ell$ sufficiently large so that $\ell > |e(T)| + (\log_2 \ell) + c$. This inequality contradicts Eq. (1).
The incompressibility method

The basic idea in this method is that “most” strings cannot be compressed very much. Generally speaking, a proof works by selecting a typical instance and arguing about its properties. In the incompressibility method, we pick a random “incompressible” string and argue about it.

Example.

Let $\pi(x)$ denote the number of primes $\leq x$.

A celebrated theorem known as the prime number theorem states that $\pi(x) \sim \frac{x}{\log x}$.

Using the incompressibility method, however, we can prove the weaker inequality $\pi(n) > \frac{cn}{(\log n)^2}$ for infinitely many $n$. 
The incompressibility method

Consider the ordinary binary representation of the non-negative integers, so that, for example, 43 is represented by 101011.

If $n \geq 1$ is represented by a string $x$, then it is easy to see that $|x| = \lfloor \log_2 n \rfloor + 1$.

Unfortunately, there are also other possible representations for 43, such as 0101011.

To avoid the “leading zeroes” problem, we can define a 1–1 mapping between the natural numbers and elements of $\{0, 1\}^*$, as follows:

e(n) is defined to be the string obtained by taking the ordinary base-2 expansion of $n + 1$, and then dropping the leading bit 1.
For example, the representations of the first 8 natural numbers are given in the following table.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$e(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>00</td>
</tr>
<tr>
<td>4</td>
<td>01</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>000</td>
</tr>
</tbody>
</table>
The incompressibility method

Note that $|e(n)| = \lfloor \log_2(n+1) \rfloor + 1 - 1 = \lfloor \log_2(n+1) \rfloor$.

Now $t + 1 \leq 2t$ for $t \geq 1$, so

$$\log_2(t + 1) \leq \log_2(2t) \leq (\log_2 t) + 1.$$  \hfill (2)

It follows that

$$(\log_2 n) - 1 \leq |e(n)| \leq (\log_2 n) + 1.$$  \hfill (3)
Previously we defined a binary string $x$ to be random if $C(x) \geq |x|$. Since we now have a bijection between binary strings and natural numbers, we can define a natural number $N$ to be \textit{random} if $C(e(N)) \geq |e(N)|$. By our theorem above, there exist infinitely many random integers.
We can use the incompressibility method to formal languages, specifically, to proving that certain languages are not regular.

*Example.* Let \( L = \{0^k1^k : k \geq 1\} \). We prove that \( L \) is not regular. Suppose it were. Then it would be accepted by a DFA \( M = (Q, \Sigma, \delta, q_0, F) \). We could then encode each integer \( n \) by providing a description of \( M \) (in \( O(1) \) bits) and \( q = \delta(q_0, 0^n) \) (in \( O(1) \) bits), because then \( n \) is uniquely specified as the least \( i \) with \( \delta(q, 1^i) \in F \). Hence \( C(e(n)) = O(1) \). But there exist infinitely many \( n \) with \( C(e(n)) \geq \log_2 n + O(1) \), a contradiction.
The incompressibility method

We can generalize the previous example, as follows:

**Lemma.** Let $L \subseteq \Sigma^*$ be regular, and define $L_x = \{y : xy \in L\}$. Then there exists a constant $c$ such that for each $x$, if $z$ is the $n$'th string in $L_x$ in lexicographic order, then $C(z) \leq C(e(n)) + c$.

**Proof.** The string $z$ can be encoded by the DFA for $L$ (in $O(1)$ bits), plus the state of the DFA after processing $x$ (in $O(1)$ bits), and the encoding $e(n)$. 
We now consider some applications of this lemma.

*Example.*

Let us prove that $L = \{1^p : p \text{ prime}\}$ is not regular.

Let $x = 1^{p_k}$, where $p_k$ is the $k$'th prime.

Then the second element of $L_x$ is $y = 1^{p_{k+1}-p_k}$.

But as $k \rightarrow \infty$, the difference $p_{k+1} - p_k$ is unbounded (because, for example, the $n - 1$ consecutive numbers $n! + 2, n! + 3, n! + 4, \ldots, n! + n$ are all composite for $n \geq 2$). Hence $C(1^{p_{k+1}-p_k})$ is unbounded.

However, by our Lemma, we have $C(y) \leq C(e(2)) + c = O(1)$, a contradiction.
Example.

Let us prove that $L = \{xx^Rw : x, w \in \{0, 1\}^+\}$ is not regular.

Let $x = (01)^m$ where $m$ is random (i. e., $C(e(m)) \geq |e(m)| \geq \log_2 m - 1$).

Then the lexicographically first element of $L_x$ is $y = (10)^m 0$. Hence $C(y) = O(1)$.

But $C(y) \geq \log_2 m + O(1)$, a contradiction.
Example.

Let us prove that $L = \{0^i1^j : \gcd(i,j) = 1\}$ is not regular.

Let $x = 0^{(p-1)!}1$ where $p$ is a prime, and $|e(p)| = n$.

Then the second word in $L_x$ is $y = 1^{p-1}$, which gives $C(y) = O(1)$.

But $C(e(p)) \leq C(y) + O(1)$, and there are infinitely many primes, so $C(e(p)) = O(1)$ for infinitely many primes $p$, a contradiction.
An open problem

Can you make the incompressibility method work for context-free languages?

This would be a big breakthrough!

The difficulty is handling the stack.

Oliver Glier (2003) has gotten it to work for DCFL’s.