We’re going to be building Turing machines that take a wide variety of different inputs: numbers, strings, lists, matrices, graphs, DFA’s, CFG’s, PDA’s, and even other Turing machines.
The need for encodings

We’re going to be building Turing machines that take a wide variety of different inputs: numbers, strings, lists, matrices, graphs, DFA’s, CFG’s, PDA’s, and even other Turing machines.

In order to do this, we have to specify exactly how these objects are going to be turned into strings that encode them.
The need for encodings

We’re going to be building Turing machines that take a wide variety of different inputs: numbers, strings, lists, matrices, graphs, DFA’s, CFG’s, PDA’s, and even other Turing machines.

In order to do this, we have to specify exactly how these objects are going to be turned into strings that encode them.

Furthermore, for a technical reason we’ll see later, we want a binary encoding using only the symbols 0 and 1.
The need for encodings

We’re going to be building Turing machines that take a wide variety of different inputs: numbers, strings, lists, matrices, graphs, DFA’s, CFG’s, PDA’s, and even other Turing machines.

In order to do this, we have to specify exactly how these objects are going to be turned into strings that encode them.

Furthermore, for a technical reason we’ll see later, we want a binary encoding using only the symbols 0 and 1.

The answer is (briefly): almost any encoding will work, provide you can decode uniquely. Every encoding should have exactly one interpretation.
Furthermore, it should be possible, given a string, to tell if it is a valid encoding or not.
The need for encodings

Furthermore, it should be possible, given a string, to tell if it is a valid encoding or not.

For example, this can be done by building a TM to decode a string, and going to the reject state if it fails.
The need for encodings

Furthermore, it should be possible, given a string, to tell if it is a valid encoding or not.

For example, this can be done by building a TM to decode a string, and going to the reject state if it fails.

Sometimes we also need to encode multiple objects at once, which we typically do by concatenating the encodings.
The need for encodings

Furthermore, it should be possible, given a string, to tell if it is a valid encoding or not.

For example, this can be done by building a TM to decode a string, and going to the reject state if it fails.

Sometimes we also need to encode multiple objects at once, which we typically do by concatenating the encodings.

In this case we demand that we can tell when one encoding ends and the other begins.
Let’s start with one of the simplest cases: encoding a single number. In this case there are two choices: we can encode a number $n$ in unary, as the string $0^n = 00\cdots0$, or in binary.
Let’s start with one of the simplest cases: encoding a single number. In this case there are two choices: we can encode a number \( n \) in \textit{unary}, as the string \( 0^n = 00 \cdots 0 \), or in binary.

Which should we choose? For our current purposes, it doesn’t matter!
Let’s start with one of the simplest cases: encoding a single number. In this case there are two choices: we can encode a number \( n \) in \textit{unary}, as the string \( 0^n = 00 \cdots 0 \), or in binary.

Which should we choose? For our current purposes, it doesn’t matter!

It doesn’t matter because there are simple Turing machines that convert from one encoding to another (exercise).
Let’s start with one of the simplest cases: encoding a single number. In this case there are two choices: we can encode a number \( n \) in *unary*, as the string \( 0^n = 00 \cdots 0 \), or in binary.

Which should we choose? For our current purposes, it doesn’t matter!

It doesn’t matter because there are simple Turing machines that convert from one encoding to another (exercise).

Here we have in mind a Turing machine that starts with one encoding on its tape, and finishes with another.
Next, let’s look at encoding strings.
Next, let’s look at encoding strings.

Here the problem is to take a string over an *given fixed alphabet*, and encode it in binary.
Next, let’s look at encoding strings.

Here the problem is to take a string over an *given fixed alphabet*, and encode it in binary.

The easiest way is with a *fixed-length encoding*. Here we replace every letter in a larger alphabet $\Sigma$ with a binary string; every letter is replaced by a string of the same length.
Next, let’s look at encoding strings.

Here the problem is to take a string over an given fixed alphabet, and encode it in binary.

The easiest way is with a fixed-length encoding. Here we replace every letter in a larger alphabet $\Sigma$ with a binary string; every letter is replaced by a string of the same length.

For example, if $\Sigma = \{a, b, c\}$, we can encode using $a \rightarrow 00$, $b \rightarrow 01$, and $c \rightarrow 10$. Then, for example, 100001 decodes to $cab$. 
One desirable property of encodings, which the above encoding does not have, is prefix-freeness: no encoding is a prefix of any other. If an encoding has this property, then it’s easy to encode lists of strings just by concatenating the encodings together.

Here is an example of a prefix-free binary encoding for strings over the alphabet Σ = {a, b, c}:

- First use the encoding we mentioned above, and then concatenate 11 on the end. So now the encoding of cab is 10000111.
- The 11 on the end serves as an “end-of-string” marker, so we can concatenate different encodings together to get concatenations of lists of strings.
Encoding strings

One desirable property of encodings, which the above encoding does not have, is prefix-freeness: no encoding is a prefix of any other. If an encoding has this property, then it’s easy to encode lists of strings just by concatenating the encodings together.

Here is an example of a prefix-free binary encoding for strings over the alphabet $\Sigma = \{a, b, c\}$: first use the encoding we mentioned above, and then concatenate 11 on the end. So now the encoding of $cab$ is 10000111.
One desirable property of encodings, which the above encoding does not have, is prefix-freeness: no encoding is a prefix of any other. If an encoding has this property, then it’s easy to encode lists of strings just by concatenating the encodings together.

Here is an example of a prefix-free binary encoding for strings over the alphabet $\Sigma = \{a, b, c\}$: first use the encoding we mentioned above, and then concatenate 11 on the end. So now the encoding of $cab$ is 10000111.

The 11 on the end serves as an “end-of-string” marker, so we can concatenate different encodings together to get concatenations of lists of strings.
Encoding strings

One desirable property of encodings, which the above encoding does not have, is prefix-freeness: no encoding is a prefix of any other. If an encoding has this property, then it’s easy to encode lists of strings just by concatenating the encodings together.

Here is an example of a prefix-free binary encoding for strings over the alphabet $\Sigma = \{a, b, c\}$: first use the encoding we mentioned above, and then concatenate 11 on the end. So now the encoding of $cab$ is 10000111.

The 11 on the end serves as an “end-of-string” marker, so we can concatenate different encodings together to get concatenations of lists of strings.

For example, what does 0100001101000011 decode to (I ask sheepishly)?
Later we’re going to need to encode strings in a subtly different setting: where the alphabet can be \textit{arbitrarily large}. 
Later we’re going to need to encode strings in a subtly different setting: where the alphabet can be *arbitrarily large*.

In this setting the alphabet $\Sigma$ is not of any bounded size, and we need a *single* encoding to handle any string over any $\Sigma$. 
Later we’re going to need to encode strings in a subtly different setting: where the alphabet can be *arbitrarily large*.

In this setting the alphabet $\Sigma$ is not of any bounded size, and we need a *single* encoding to handle any string over any $\Sigma$.

At first glance this seems impossible, but we use a trick.
Encoding strings over an arbitrary alphabet

Later we’re going to need to encode strings in a subtly different setting: where the alphabet can be arbitrarily large.

In this setting the alphabet $\Sigma$ is not of any bounded size, and we need a single encoding to handle any string over any $\Sigma$.

At first glance this seems impossible, but we use a trick.

We make one infinite alphabet that contains all possible symbols.
You can think of this as a super-duper form of ASCII or UNICODE where instead of slots for 128 or 65536 symbols we have slots for infinitely many different symbols.
Encoding strings over an arbitrary alphabet

You can think of this as a super-duper form of ASCII or UNICODE where instead of slots for 128 or 65536 symbols we have slots for infinitely many different symbols.

Let us say the super-duper alphabet is \( \{ a_1, a_2, a_3, \ldots \} \). Every finite alphabet we talk about will be a subset of this one.
Encoding strings over an arbitrary alphabet

You can think of this as a super-duper form of ASCII or UNICODE where instead of slots for 128 or 65536 symbols we have slots for infinitely many different symbols.

Let us say the super-duper alphabet is \{a_1, a_2, a_3, \ldots\}. Every finite alphabet we talk about will be a subset of this one.

Then we can encode \(a_i\) by the binary string 0\(^i\)1.
Encoding strings over an arbitrary alphabet

You can think of this as a super-duper form of ASCII or UNICODE where instead of slots for 128 or 65536 symbols we have slots for infinitely many different symbols.

Let us say the super-duper alphabet is \( \{a_1, a_2, a_3, \ldots \} \). Every finite alphabet we talk about will be a subset of this one.

Then we can encode \( a_i \) by the binary string 0\( i \)1.

Using this idea we have a uniform way to encode any string over any alphabet.
Now suppose we want to encode lists of natural numbers (like vectors), or sets.
Encoding lists (sets) of numbers

Now suppose we want to encode lists of natural numbers (like vectors), or sets.

If we use binary encoding of numbers, then we can’t just concatenate the encodings if we want unique decoding.

Exercise: how would you encode lists of arbitrary integers, allowing negative numbers?
Now suppose we want to encode lists of natural numbers (like vectors), or sets.

If we use binary encoding of numbers, then we can’t just concatenate the encodings if we want unique decoding.

So instead we can do it in two steps. First encode each number in binary, using $a$ for 0 and $b$ for 1. Then separate the numbers with a delimiter character like $c$. Now we have a string over the alphabet $\Sigma = \{a, b, c\}$, which we can encode again in a prefix-free way using the method of a previous slide.
Now suppose we want to encode lists of natural numbers (like vectors), or sets.

If we use binary encoding of numbers, then we can’t just concatenate the encodings if we want unique decoding.

So instead we can do it in two steps. First encode each number in binary, using $a$ for 0 and $b$ for 1. Then separate the numbers with a delimiter character like $c$. Now we have a string over the alphabet $\Sigma = \{a, b, c\}$, which we can encode again in a prefix-free way using the method of a previous slide.

So for example the vector (or list) (2, 0, 21) would be encoded first as $bacacbababc$ and then as 0100100100100100011011.
Encoding lists (sets) of numbers

Now suppose we want to encode lists of natural numbers (like vectors), or sets.

If we use binary encoding of numbers, then we can’t just concatenate the encodings if we want unique decoding.

So instead we can do it in two steps. First encode each number in binary, using $a$ for 0 and $b$ for 1. Then separate the numbers with a delimiter character like $c$. Now we have a string over the alphabet $\Sigma = \{a, b, c\}$, which we can encode again in a prefix-free way using the method of a previous slide.

So for example the vector (or list) $(2, 0, 21)$ would be encoded first as $bacacbabc$ and then as $010010001001000100011011$.

Exercise: how would you encode lists of arbitrary integers, allowing negative numbers?
How do we encode matrices of natural numbers? This seems harder because a matrix is fundamentally a two-dimensional object.

One way to do it is as follows: write numbers in binary, use brackets to delimit each row, and # between the numbers. This gives us an encoding over the alphabet \{0, 1, [ , ] , #\}, which we can then turn into a binary encoding using the ideas before.

So in a first step the matrix

\[
\begin{pmatrix}
3 & 4 & 5 \\
1 & 0 & 2 \\
1 & 1 & 3 \\
\end{pmatrix}
\]

becomes \[[11#100#101][1#0#10][1011#11#1]\] which is then encoded further into binary.

This easily extends to 3-dimensional arrays, etc.
How do we encode matrices of natural numbers? This seems harder because a matrix is fundamentally a two-dimensional object.

One way to do it is as follows: write numbers in binary, use brackets to delimit each row, and \# between the numbers. This gives us an encoding over the alphabet \{0, 1, [, ], \#\}, which we can then turn into a binary encoding using the ideas before.

So in a first step the matrix

```
3 4 5
1 0 2
11 3 1
```

becomes \[
[11\#100\#101]
[1\#0\#10]
[1011\#11\#1]
\] which is then encoded further into binary.

This easily extends to 3-dimensional arrays, etc.
Encoding matrices

How do we encode matrices of natural numbers? This seems harder because a matrix is fundamentally a two-dimensional object.

One way to do it is as follows: write numbers in binary, use brackets to delimit each row, and # between the numbers. This gives us an encoding over the alphabet \{0, 1, [, ], #\}, which we can then turn into a binary encoding using the ideas before.

So in a first step the matrix

\[
\begin{array}{ccc}
3 & 4 & 5 \\
1 & 0 & 2 \\
11 & 3 & 1 \\
\end{array}
\]

becomes \[[11#100#11][1#0#10][1011#11#1]\] which is then encoded further into binary.
How do we encode matrices of natural numbers? This seems harder because a matrix is fundamentally a two-dimensional object.

One way to do it is as follows: write numbers in binary, use brackets to delimit each row, and \# between the numbers. This gives us an encoding over the alphabet \{0, 1, [ , ] , \#\}, which we can then turn into a binary encoding using the ideas before.

So in a first step the matrix

\[
\begin{pmatrix}
3 & 4 & 5 \\
1 & 0 & 2 \\
11 & 3 & 1 \\
\end{pmatrix}
\]

becomes \[[11\#100\#101][1\#0\#10][1011\#11\#1]]\] which is then encoded further into binary.

This easily extends to 3-dimensional arrays, etc.
Now let’s turn to encoding graphs.
Now let’s turn to encoding graphs.

A graph consists of a list of vertices $V$ and a list of edges $E$. 
Now let’s turn to encoding graphs.

A graph consists of a list of vertices \( V \) and a list of edges \( E \).

We can encode each vertex as a string or number, and each edge as a pair (ordered if it is a directed graph).
Now let’s turn to encoding graphs.

A graph consists of a list of vertices $V$ and a list of edges $E$.

We can encode each vertex as a string or number, and each edge as a pair (ordered if it is a directed graph).

Then we can use our ideas for encoding lists as above.
Suppose we have a function $f : S \rightarrow T$, where $S$ and $T$ are finite sets, and we have a way to encode the elements of $S$ and $T$. Let the binary encoding of $x \in S \cup T$ be written $e(x)$. Let $S = \{x_1, x_2, \ldots, x_n\}$. Then we can encode $f$ by first creating the string $e(x_1)\#e(f(x_1))\#e(x_2)\#e(f(x_2))\cdots e(x_n)\#e(f(x_n))$ and then encoding this string (which uses the alphabet $\{0, 1, \#\}$) as we did before on a previous slide.
Suppose we have a function $f : S \rightarrow T$, where $S$ and $T$ are finite sets, and we have a way to encode the elements of $S$ and $T$.

Let the binary encoding of $x \in S \cup T$ be written $e(x)$. 
Suppose we have a function $f : S \rightarrow T$, where $S$ and $T$ are finite sets, and we have a way to encode the elements of $S$ and $T$.

Let the binary encoding of $x \in S \cup T$ be written $e(x)$.

Let $S = \{x_1, x_2, \ldots, x_n\}$. 
Suppose we have a function \( f : S \rightarrow T \), where \( S \) and \( T \) are finite sets, and we have a way to encode the elements of \( S \) and \( T \).

Let the binary encoding of \( x \in S \cup T \) be written \( e(x) \).

Let \( S = \{x_1, x_2, \ldots, x_n\} \).

Then we can encode \( f \) by first creating the string

\[
e(x_1)\#e(f(x_1))\#e(x_2)\#e(f(x_2))\cdots e(x_n)\#e(f(x_n))\
\]

and then encoding this string (which uses the alphabet \( \{0, 1, \#\} \)) as we did before on a previous slide.
A DFA consists of five things: $Q$, $\Sigma$, $\delta$, $q_0$, and $F$. 
A DFA consists of five things: $Q$, $\Sigma$, $\delta$, $q_0$, and $F$.

We can encode each of these separately as binary strings, as above (for example, $\delta$ is a finite function).
A DFA consists of five things: $Q$, $\Sigma$, $\delta$, $q_0$, and $F$.

We can encode each of these separately as binary strings, as above (for example, $\delta$ is a finite function).

Then we can combine these binary encodings using $\#$ as a delimiter.
A DFA consists of five things: $Q$, $\Sigma$, $\delta$, $q_0$, and $F$.

We can encode each of these separately as binary strings, as above (for example, $\delta$ is a finite function).

Then we can combine these binary encodings using $#$ as a delimiter.

Finally, we re-encode using our previous scheme.
Exercise: how would you encode a CFG?