Computability

What we've been studying is often called *computability theory*.
We've learned that:

Computability

What we've been studying is often called *computability theory*.
We've learned that:

- Regular Languages can be generated by Deterministic Regular Grammars, and Regular Languages can be recognized by Deterministic Finite Automata.

Computability

What we've been studying is often called *computability theory*.
We've learned that:

- Regular Languages can be generated by Deterministic Regular Grammars, and Regular Languages can be recognized by Deterministic Finite Automata.
- Non-determinism in regular grammars and in finite automata does not change what is computable.

Computability

What we've been studying is often called *computability theory*.
We've learned that:

- Regular Languages can be generated by Deterministic Regular Grammars, and Regular Languages can be recognized by Deterministic Finite Automata.
- Non-determinism in regular grammars and in finite automata does not change what is computable. (That is, they generate and recognize the same class of languages.)

Computability

What we've been studying is often called *computability theory*.
We've learned that:

- Regular Languages can be generated by Deterministic Regular Grammars, and Regular Languages can be recognized by Deterministic Finite Automata.
- Non-determinism in regular grammars and in finite automata does not change what is computable. (That is, they generate and recognize the same class of languages.)
- Context Free Languages are generated by Context Free Grammars and Context Free Languages are recognized by NPDAs.

Computability

What we've been studying is often called *computability theory*.
We've learned that:

- Regular Languages can be generated by Deterministic Regular Grammars, and Regular Languages can be recognized by Deterministic Finite Automata.
- Non-determinism in regular grammars and in finite automata does not change what is computable. (That is, they generate and recognize the same class of languages.)
- Context Free Languages are generated by Context Free Grammars and Context Free Languages are recognized by NPDAs.
- Context Free Languages are a super-set of the Regular Languages:
  There are some languages that are computable by CFGs and NPDAs, but not computable by Regular Grammars and DFAs.

Computability

What we've been studying is often called *computability theory*.
We've learned that:

- Regular Languages can be generated by Deterministic Regular Grammars, and Regular Languages can be recognized by Deterministic Finite Automata.
- Non-determinism in regular grammars and in finite automata does not change what is computable. (That is, they generate and recognize the same class of languages.)
- Context Free Languages are generated by Context Free Grammars and Context Free Languages are recognized by NPDAs.
- Context Free Languages are a super-set of the Regular Languages: There are some languages that are computable by CFGs and NPDAs, but not computable by Regular Grammars and DFAs.
- Non-Determinism *does* impact what is computable when modeling CFLs: There is a CFL that is computable by an NPDA, but not by a DPDA.

Computability

What we've been studying is often called *computability theory*.
We've learned that:

- Regular Languages can be generated by Deterministic Regular Grammars, and Regular Languages can be recognized by Deterministic Finite Automata.
- Non-determinism in regular grammars and in finite automata does not change what is computable. (That is, they generate and recognize the same class of languages.)
- Context Free Languages are generated by Context Free Grammars and Context Free Languages are recognized by NPDAs.
- Context Free Languages are a super-set of the Regular Languages: There are some languages that are computable by CFGs and NPDAs, but not computable by Regular Grammars and DFAs.
- Non-Determinism *does* impact what is computable when modeling CFLs: There is a CFL that is computable by an NPDA, but not by a DPDA.
- There exist languages that are not computable by DFGs and NPDAs.

Computability

What we've been studying is often called *computability theory*.
We've learned that:

- Regular Languages can be generated by Deterministic Regular Grammars, and Regular Languages can be recognized by Deterministic Finite Automata.
- Non-determinism in regular grammars and in finite automata does not change what is computable. (That is, they generate and recognize the same class of languages.)
- Context Free Languages are generated by Context Free Grammars and Context Free Languages are recognized by NPDAs.
- Context Free Languages are a super-set of the Regular Languages: There are some languages that are computable by CFGs and NPDAs, but not computable by Regular Grammars and DFAs.
- Non-Determinism *does* impact what is computable when modeling CFLs: There is a CFL that is computable by an NPDA, but not by a DPDA.
- There exist languages that are not computable by DFGs and NPDAs.

Still to come:

Computability

What we've been studying is often called *computability theory*.
We've learned that:

- Regular Languages can be generated by Deterministic Regular Grammars, and Regular Languages can be recognized by Deterministic Finite Automata.
- Non-determinism in regular grammars and in finite automata does not change what is computable. (That is, they generate and recognize the same class of languages.)
- Context Free Languages are generated by Context Free Grammars and Context Free Languages are recognized by NPDAs.
- Context Free Languages are a super-set of the Regular Languages:
  There are some languages that are computable by CFGs and NPDAs, but not computable by Regular Grammars and DFAs.
- Non-Determinism *does* impact what is computable when modeling CFLs:
  There is a CFL that is computable by an NPDA, but not by a DPDA.
- There exist languages that are not computable by DFGs and NPDAs.

Still to come:

- Some of these languages are computable by *Turing Machines*

Computability

What we've been studying is often called *computability theory*.
We've learned that:

- Regular Languages can be generated by Deterministic Regular Grammars, and Regular Languages can be recognized by Deterministic Finite Automata.
- Non-determinism in regular grammars and in finite automata does not change what is computable. (That is, they generate and recognize the same class of languages.)
- Context Free Languages are generated by Context Free Grammars and Context Free Languages are recognized by NPDAs.
- Context Free Languages are a super-set of the Regular Languages: There are some languages that are computable by CFGs and NPDAs, but not computable by Regular Grammars and DFAs.
- Non-Determinism *does* impact what is computable when modeling CFLs: There is a CFL that is computable by an NPDA, but not by a DPDA.
- There exist languages that are not computable by DFGs and NPDAs.

Still to come:

- Some of these languages are computable by *Turing Machines*
- Some languages are not computable by **any** machine!!!!

Computability

What we've been studying is often called *computability theory*.
We've learned that:

- Regular Languages can be generated by Deterministic Regular Grammars, and Regular Languages can be recognized by Deterministic Finite Automata.
- Non-determinism in regular grammars and in finite automata does not change what is computable. (That is, they generate and recognize the same class of languages.)
- Context Free Languages are generated by Context Free Grammars and Context Free Languages are recognized by NPDAs.
- Context Free Languages are a super-set of the Regular Languages: There are some languages that are computable by CFGs and NPDAs, but not computable by Regular Grammars and DFAs.
- Non-Determinism *does* impact what is computable when modeling CFLs: There is a CFL that is computable by an NPDA, but not by a DPDA.
- There exist languages that are not computable by DFGs and NPDAs.

Still to come:

- Some of these languages are computable by *Turing Machines*
- Some languages are not computable by **any** machine!!!! (Wow.)

Turing Machines

Invented by Alan Turing in the 1930's to try and capture what tasks are computable.

Turing Machines

Invented by Alan Turing in the 1930's to try and capture what tasks are computable.
Believed to be as powerful as *any* model of computation.

Turing Machines

Invented by Alan Turing in the 1930's to try and capture what tasks are computable.
Believed to be as powerful as *any* model of computation.
Certainly as powerful as any computer you use.

Turing Machines

Invented by Alan Turing in the 1930's to try and capture what tasks are computable.
Believed to be as powerful as *any* model of computation.
Certainly as powerful as any computer you use.

What about quantum computers?

Turing Machines

Invented by Alan Turing in the 1930's to try and capture what tasks are computable.
Believed to be as powerful as *any* model of computation.
Certainly as powerful as any computer you use.

What about quantum computers?

- They seem to make certain computations faster

Turing Machines

Invented by Alan Turing in the 1930's to try and capture what tasks are computable.
Believed to be as powerful as *any* model of computation.
Certainly as powerful as any computer you use.

What about quantum computers?

- They seem to make certain computations faster
- We have limited proof that the same speedups are *impossible* on classical machines.

Turing Machines

Invented by Alan Turing in the 1930's to try and capture what tasks are computable.
Believed to be as powerful as *any* model of computation.
Certainly as powerful as any computer you use.

What about quantum computers?

- They seem to make certain computations faster
- We have limited proof that the same speedups are *impossible* on classical machines.
- They do not change what is *computable.*

Turing Machines

Still a state machine. Two relaxations:

Turing Machines

Still a state machine. Two relaxations:

- The input isn't read-once anymore. Read/Write, and can be repeatedly accessed.

Turing Machines

Still a state machine. Two relaxations:

- The input isn't read-once anymore. Read/Write, and can be repeatedly accessed.
- The memory isn't constrained to be a stack.

Turing Machines

Still a state machine. Two relaxations:

- The input isn't read-once anymore. Read/Write, and can be repeatedly accessed.
- The memory isn't constrained to be a stack.

Input starts out on an Infinite "tape" of cells, one character per cell:
$\Delta, x_1, x_2, \ldots, x_n, \Delta, \Delta, \ldots$

Turing Machines

Still a state machine. Two relaxations:

- The input isn't read-once anymore. Read/Write, and can be repeatedly accessed.
- The memory isn't constrained to be a stack.

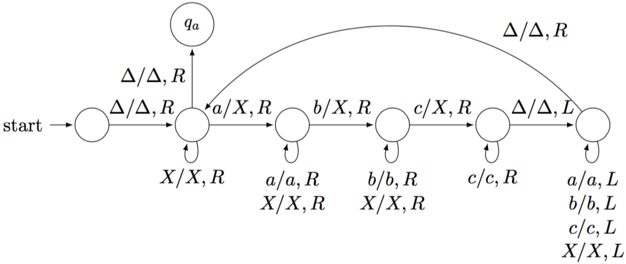
Input starts out on an Infinite "tape" of cells, one character per cell:
$\Delta, x_1, x_2, \ldots, x_n, \Delta, \Delta, \ldots$
The "tape head" (pointer?) starts at the left-most point, the Δ before the input.

Turing Machines

Still a state machine. Two relaxations:

- The input isn't read-once anymore. Read/Write, and can be repeatedly accessed.
- The memory isn't constrained to be a stack.

Input starts out on an Infinite "tape" of cells, one character per cell:
$\Delta, x_1, x_2, \ldots, x_n, \Delta, \Delta, \ldots$
The "tape head" (pointer?) starts at the left-most point, the Δ before the input.



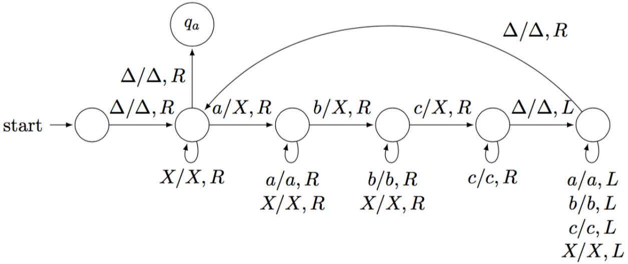
During each transition, write a new symbol, and move the pointer 1 cell (R or L).

Turing Machines

Still a state machine. Two relaxations:

- The input isn't read-once anymore. Read/Write, and can be repeatedly accessed.
- The memory isn't constrained to be a stack.

Input starts out on an Infinite "tape" of cells, one character per cell:
$\Delta, x_1, x_2, \ldots, x_n, \Delta, \Delta, \ldots$
The "tape head" (pointer?) starts at the left-most point, the Δ before the input.



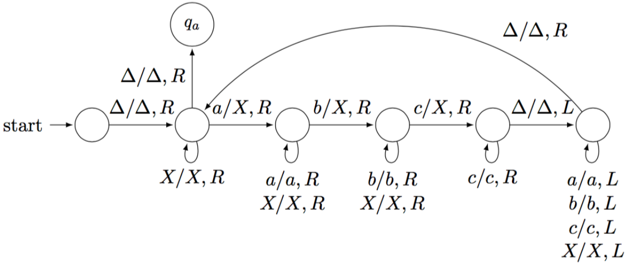
During each transition, write a new symbol, and move the pointer 1 cell (R or L).
A string is accepted if it ends in a special (sink) accept state.

Turing Machines

Still a state machine. Two relaxations:

- The input isn't read-once anymore. Read/Write, and can be repeatedly accessed.
- The memory isn't constrained to be a stack.

Input starts out on an Infinite "tape" of cells, one character per cell:
$\Delta, x_1, x_2, \ldots, x_n, \Delta, \Delta, \ldots$
The "tape head" (pointer?) starts at the left-most point, the Δ before the input.



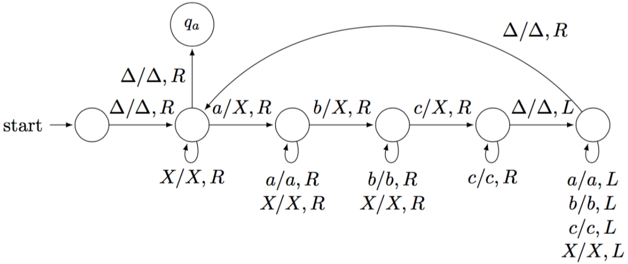
During each transition, write a new symbol, and move the pointer 1 cell (R or L).
A string is accepted if it ends in a special (sink) accept state.
A string is rejected if the machine halts in a special reject state,

Turing Machines

Still a state machine. Two relaxations:

- The input isn't read-once anymore. Read/Write, and can be repeatedly accessed.
- The memory isn't constrained to be a stack.

Input starts out on an Infinite "tape" of cells, one character per cell:
$\Delta, x_1, x_2, \ldots, x_n, \Delta, \Delta, \ldots$
The "tape head" (pointer?) starts at the left-most point, the Δ before the input.



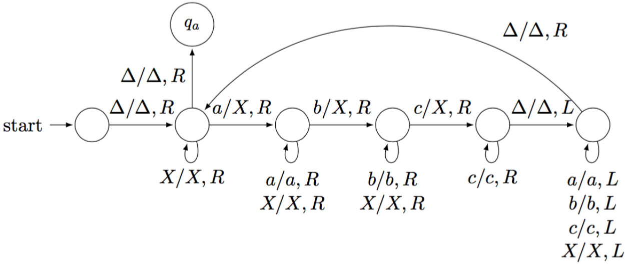
During each transition, write a new symbol, and move the pointer 1 cell (R or L).
A string is accepted if it ends in a special (sink) accept state.
A string is rejected if the machine halts in a special reject state, or if it runs forever!

Turing Machines

Still a state machine. Two relaxations:

- The input isn't read-once anymore. Read/Write, and can be repeatedly accessed.
- The memory isn't constrained to be a stack.

Input starts out on an Infinite "tape" of cells, one character per cell:
$\Delta, x_1, x_2, \ldots, x_n, \Delta, \Delta, \ldots$
The "tape head" (pointer?) starts at the left-most point, the Δ before the input.



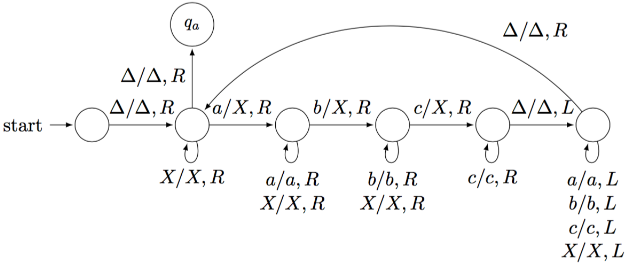
During each transition, write a new symbol, and move the pointer 1 cell (R or L).
A string is accepted if it ends in a special (sink) accept state.
A string is rejected if the machine halts in a special reject state, or if it runs forever!
If a TM halts on every input, we say it *decides* the language that it accepts.

Turing Machines

Still a state machine. Two relaxations:

- The input isn't read-once anymore. Read/Write, and can be repeatedly accessed.
- The memory isn't constrained to be a stack.

Input starts out on an Infinite "tape" of cells, one character per cell:
$\Delta, x_1, x_2, \ldots, x_n, \Delta, \Delta, \ldots$
The "tape head" (pointer?) starts at the left-most point, the Δ before the input.



During each transition, write a new symbol, and move the pointer 1 cell (R or L).
A string is accepted if it ends in a special (sink) accept state.
A string is rejected if the machine halts in a special reject state, or if it runs forever!
If a TM halts on every input, we say it *decides* the language that it accepts.
If a TM runs for ever on some input, we say it *recognizes* the language that it accepts.

Computability

Nondeterminism does impact what is computable on a Turing Machine.

Computability

Nondeterminism does impact what is computable on a Turing Machine.
Imagine L is decided by a non-deterministic TM, M.
You can construct a deterministic TM that simulates M, trying every possible choice.

Computability

Nondeterminism does impact what is computable on a Turing Machine.
Imagine L is decided by a non-deterministic TM, M.
You can construct a deterministic TM that simulates M, trying every possible choice.

Time complexity:
How many transitions does our TM require?

Computability

Nondeterminism does impact what is computable on a Turing Machine.
Imagine L is decided by a non-deterministic TM, M.
You can construct a deterministic TM that simulates M, trying every possible choice.

Time complexity:
How many transitions does our TM require?
We measure runtime as a function: the number of transitions, as a function of the input size.

Computability

Nondeterminism does impact what is computable on a Turing Machine.
Imagine L is decided by a non-deterministic TM, M.
You can construct a deterministic TM that simulates M, trying every possible choice.

Time complexity:
How many transitions does our TM require?
We measure runtime as a function: the number of transitions, as a function of the input size.
How many transitions did we need for $a^n b^n c^n$?

Computability

Nondeterminism does impact what is computable on a Turing Machine.
Imagine L is decided by a non-deterministic TM, M.
You can construct a deterministic TM that simulates M, trying every possible choice.

Time complexity:
How many transitions does our TM require?
We measure runtime as a function: the number of transitions, as a function of the input size.
How many transitions did we need for $a^n b^n c^n$?

Known: Random Access Memory can improve runtime. (Think of Binary search)

Computability

Nondeterminism does impact what is computable on a Turing Machine.
Imagine L is decided by a non-deterministic TM, M.
You can construct a deterministic TM that simulates M, trying every possible choice.

Time complexity:
How many transitions does our TM require?
We measure runtime as a function: the number of transitions, as a function of the input size.
How many transitions did we need for $a^n b^n c^n$?

Known: Random Access Memory can improve runtime. (Think of Binary search)
Conjectured: Quantum computation can improve runtime.

Computability

Nondeterminism does impact what is computable on a Turing Machine.
Imagine L is decided by a non-deterministic TM, M.
You can construct a deterministic TM that simulates M, trying every possible choice.

Time complexity:
How many transitions does our TM require?
We measure runtime as a function: the number of transitions, as a function of the input size.
How many transitions did we need for $a^n b^n c^n$?

Known: Random Access Memory can improve runtime. (Think of Binary search)
Conjectured: Quantum computation can improve runtime.
Conjectured: Non-determinism can improve runtime:
$\mathcal{P}$ is the class of languages decidable in polynomial time on a deterministic TM.
$\mathcal{NP}$ is the class of languages decidable in polynomial time on a non-deterministic TM.

Computability

Nondeterminism does impact what is computable on a Turing Machine.
Imagine L is decided by a non-deterministic TM, M.
You can construct a deterministic TM that simulates M, trying every possible choice.

Time complexity:
How many transitions does our TM require?
We measure runtime as a function: the number of transitions, as a function of the input size.
How many transitions did we need for $a^n b^n c^n$?

Known: Random Access Memory can improve runtime. (Think of Binary search)
Conjectured: Quantum computation can improve runtime.
Conjectured: Non-determinism can improve runtime:
$\mathcal{P}$ is the class of languages decidable in polynomial time on a deterministic TM.
$\mathcal{NP}$ is the class of languages decidable in polynomial time on a non-deterministic TM.

$$\mathcal{P} \neq \mathcal{NP}?$$

Since 2000, there has been a $1 Million prize offered for proving the conjecture.

Formalizing and Serializing TMs

A TM is a 7-tuple: $M = (Q, \Sigma, \Gamma, q_0, q_a, q_r, \delta)$, where:

Formalizing and Serializing TMs

A TM is a 7-tuple: $M = (Q, \Sigma, \Gamma, q_0, q_a, q_r, \delta)$, where:

- Q is the finite set of states.
- Σ is the input alphabet, and $\Delta \notin \Sigma$.
- Γ is the tape alphabet, $\Sigma \subset \Gamma$ and $\Delta \in \Gamma$.

Formalizing and Serializing TMs

A TM is a 7-tuple: $M = (Q, \Sigma, \Gamma, q_0, q_a, q_r, \delta)$, where:

- Q is the finite set of states.
- Σ is the input alphabet, and $\Delta \notin \Sigma$.
- Γ is the tape alphabet, $\Sigma \subset \Gamma$ and $\Delta \in \Gamma$.
- $q_0 \in Q$ is the start state.
- $q_a \in Q$ is the accept state.
- $q_r \in Q$ is the reject state.

Formalizing and Serializing TMs

A TM is a 7-tuple: $M = (Q, \Sigma, \Gamma, q_0, q_a, q_r, \delta)$, where:

- Q is the finite set of states.
- Σ is the input alphabet, and $\Delta \notin \Sigma$.
- Γ is the tape alphabet, $\Sigma \subset \Gamma$ and $\Delta \in \Gamma$.
- $q_0 \in Q$ is the start state.
- $q_a \in Q$ is the accept state.
- $q_r \in Q$ is the reject state.
- δ is the transition function, $\delta : Q \setminus \{q_a, q_r\} \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$

Formalizing and Serializing TMs

A TM is a 7-tuple: $M = (Q, \Sigma, \Gamma, q_0, q_a, q_r, \delta)$, where:

- Q is the finite set of states.
- Σ is the input alphabet, and $\Delta \notin \Sigma$.
- Γ is the tape alphabet, $\Sigma \subset \Gamma$ and $\Delta \in \Gamma$.
- $q_0 \in Q$ is the start state.
- $q_a \in Q$ is the accept state.
- $q_r \in Q$ is the reject state.
- δ is the transition function, $\delta : Q \setminus \{q_a, q_r\} \times \Gamma \to Q \times \Gamma \times \{L, R\}$

What does it take to write down the description of a TM?

Formalizing and Serializing TMs

A TM is a 7-tuple: $M = (Q, \Sigma, \Gamma, q_0, q_a, q_r, \delta)$, where:

- Q is the finite set of states.
- Σ is the input alphabet, and $\Delta \notin \Sigma$.
- Γ is the tape alphabet, $\Sigma \subset \Gamma$ and $\Delta \in \Gamma$.
- $q_0 \in Q$ is the start state.
- $q_a \in Q$ is the accept state.
- $q_r \in Q$ is the reject state.
- δ is the transition function, $\delta : Q \setminus \{q_a, q_r\} \times \Gamma \to Q \times \Gamma \times \{L, R\}$

What does it take to write down the description of a TM?
We can represent the description using a binary string.

Formalizing and Serializing TMs

A TM is a 7-tuple: $M = (Q, \Sigma, \Gamma, q_0, q_a, q_r, \delta)$, where:

- Q is the finite set of states.
- Σ is the input alphabet, and $\Delta \notin \Sigma$.
- Γ is the tape alphabet, $\Sigma \subset \Gamma$ and $\Delta \in \Gamma$.
- $q_0 \in Q$ is the start state.
- $q_a \in Q$ is the accept state.
- $q_r \in Q$ is the reject state.
- δ is the transition function, $\delta : Q \setminus \{q_a, q_r\} \times \Gamma \to Q \times \Gamma \times \{L, R\}$

What does it take to write down the description of a TM?
We can represent the description using a binary string.
Lot's of ways to do this, but here is one:

Formalizing and Serializing TMs

A TM is a 7-tuple: $M = (Q, \Sigma, \Gamma, q_0, q_a, q_r, \delta)$, where:

- Q is the finite set of states.
- Σ is the input alphabet, and $\Delta \notin \Sigma$.
- Γ is the tape alphabet, $\Sigma \subset \Gamma$ and $\Delta \in \Gamma$.
- $q_0 \in Q$ is the start state.
- $q_a \in Q$ is the accept state.
- $q_r \in Q$ is the reject state.
- δ is the transition function, $\delta : Q \setminus \{q_a, q_r\} \times \Gamma \to Q \times \Gamma \times \{L, R\}$

What does it take to write down the description of a TM?
We can represent the description using a binary string.
Lot's of ways to do this, but here is one:
$Q = \{q_1, q_2, q_3, q_4\}$ can be written as 0101101110111110.

Formalizing and Serializing TMs

A TM is a 7-tuple: $M = (Q, \Sigma, \Gamma, q_0, q_a, q_r, \delta)$, where:

- Q is the finite set of states.
- Σ is the input alphabet, and $\Delta \notin \Sigma$.
- Γ is the tape alphabet, $\Sigma \subset \Gamma$ and $\Delta \in \Gamma$.
- $q_0 \in Q$ is the start state.
- $q_a \in Q$ is the accept state.
- $q_r \in Q$ is the reject state.
- δ is the transition function, $\delta : Q \setminus \{q_a, q_r\} \times \Gamma \to Q \times \Gamma \times \{L, R\}$

What does it take to write down the description of a TM?
We can represent the description using a binary string.
Lot's of ways to do this, but here is one:
$Q = \{q_1, q_2, q_3, q_4\}$ can be written as 010110111011110.
$\Sigma = \{a, b, c\}$ can be written as 0101101110.

Formalizing and Serializing TMs

A TM is a 7-tuple: $M = (Q, \Sigma, \Gamma, q_0, q_a, q_r, \delta)$, where:

- Q is the finite set of states.
- Σ is the input alphabet, and $\Delta \notin \Sigma$.
- Γ is the tape alphabet, $\Sigma \subset \Gamma$ and $\Delta \in \Gamma$.
- $q_0 \in Q$ is the start state.
- $q_a \in Q$ is the accept state.
- $q_r \in Q$ is the reject state.
- δ is the transition function, $\delta : Q \setminus \{q_a, q_r\} \times \Gamma \to Q \times \Gamma \times \{L, R\}$

What does it take to write down the description of a TM?
We can represent the description using a binary string.
Lot's of ways to do this, but here is one:
$Q = \{q_1, q_2, q_3, q_4\}$ can be written as 0101101110111110.
$\Sigma = \{a, b, c\}$ can be written as 0101101110.
We will talk about the descriptions of machines.
When we talk about the binary string representing machine M, we will write $\langle M \rangle$.

Formalizing and Serializing TMs

A TM is a 7-tuple: $M = (Q, \Sigma, \Gamma, q_0, q_a, q_r, \delta)$, where:

- Q is the finite set of states.
- Σ is the input alphabet, and $\Delta \notin \Sigma$.
- Γ is the tape alphabet, $\Sigma \subset \Gamma$ and $\Delta \in \Gamma$.
- $q_0 \in Q$ is the start state.
- $q_a \in Q$ is the accept state.
- $q_r \in Q$ is the reject state.
- δ is the transition function, $\delta : Q \setminus \{q_a, q_r\} \times \Gamma \to Q \times \Gamma \times \{L, R\}$

What does it take to write down the description of a TM?
We can represent the description using a binary string.
Lot's of ways to do this, but here is one:
$Q = \{q_1, q_2, q_3, q_4\}$ can be written as 0101101110111110.
$\Sigma = \{a, b, c\}$ can be written as 0101101110.
We will talk about the descriptions of machines.
When we talk about the binary string representing machine M, we will write $\langle M \rangle$.

We can build a universal TM that recognizes the following language:
$L_U = \{\langle M \rangle 0x \mid \text{ TM } M \text{ accepts input } x\}$