In these slides Dana Angluin’s learning algorithm for DFA’s is explained.
In these slides Dana Angluin’s learning algorithm for DFA’s is explained.

In this learning model the learning algorithm has access to a teacher answering the following types of queries:

- **Membership Queries**: On a membership query, the learning algorithm selects a string $s$ and asks the teacher if $s$ is in the language that the learning algorithm is attempting to learn.

- **Equivalence Queries**: On an equivalence query, the learning algorithm submits a DFA $\hat{M}$ and asks the teacher if $\hat{M}$ recognizes the language under consideration. If $\hat{M}$ is not correct, the teacher provides a counterexample. A counterexample is a string that $\hat{M}$ should accept, but which is rejected by $\hat{M}$, or vice versa.
In these slides Dana Angluin’s learning algorithm for DFA’s is explained.

In this learning model the learning algorithm has access to a teacher answering the following types of queries:

- **Membership Queries:** On a membership query, the learning algorithm selects a string $s$ and asks the teacher if $s$ is in the language that the learning algorithm is attempting to learn.

- **Equivalence Queries:** On an equivalence query, the learning algorithm submits a DFA $\hat{M}$ and asks the teacher if $\hat{M}$ recognizes the language under consideration. If $\hat{M}$ is not correct, the teacher provides a counterexample. A counterexample is a string that $\hat{M}$ should accept, but which is rejected by $\hat{M}$, or vice versa.
In these slides Dana Angluin’s learning algorithm for DFA’s is explained.

In this learning model the learning algorithm has access to a teacher answering the following types of queries:

- **Membership Queries:** On a membership query, the learning algorithm selects a string $s$ and ask the teacher if $s$ is in the language that the learning algorithm is attempting to learn.

- **Equivalence Queries:** On an equivalence query, the learning algorithm submits a DFA $\hat{M}$ and asks the teacher if $\hat{M}$ recognises the language under consideration. If $\hat{M}$ is not correct, the teacher provides a counterexample.
In these slides Dana Angluin’s learning algorithm for DFA’s is explained.

In this learning model the learning algorithm has access to a teacher answering the following types of queries:

- **Membership Queries:** On a membership query, the learning algorithm selects a string $s$ and ask the teacher if $s$ is in the language that the learning algorithm is attempting to learn.

- **Equivalence Queries:** On an equivalence query, the learning algorithm submits a DFA $\hat{M}$ and asks the teacher if $\hat{M}$ recognises the language under consideration. If $\hat{M}$ is not correct, the teacher provides a counterexample. A counterexample is a string that $\hat{M}$ should accept, but which is rejected by $\hat{M}$, or vice versa.
Exact Learning of Finite Automata

Let $M$ be the target automaton, and assume $M$ is minimal. $\text{size}(M)$ is the number of states in $M$. 
Let \( M \) be the target automaton, and assume \( M \) is minimal. \( \text{size}(M) \) is the number of states in \( M \).

The learner attempts to continually discover new states of \( M \). A new state is a state exhibiting behavior that is demonstrably different from the states discovered so far.
Let $M$ be the target automaton, and assume $M$ is minimal. $\text{size}(M)$ is the number of states in $M$.

The learner attempts to continually discover new states of $M$. A new state is a state exhibiting behavior that is demonstrably different from the states discovered so far.

The learner runs in phases. In each phase, the learner constructs a hypothesis automaton $\hat{M}$ with states the currently discovered states of $M$. The learning algorithm then makes an equivalence query. This query asks if $M$ and $\hat{M}$ are equivalent. The counterexample from this equivalence query allows the algorithm to use membership queries to discover a new state of $M$. When all the states of $M$ have been discovered, $\hat{M} = M$. 
Let $M$ be the target automaton, and assume $M$ is minimal. $\text{size}(M)$ is the number of states in $M$.

The learner attempts to continually discover new states of $M$. A new state is a state exhibiting behavior that is demonstrably different from the states discovered so far.

The learner runs in phases. In each phase, the learner constructs a hypothesis automaton $\hat{M}$ with states the currently discovered states of $M$.

The learning algorithm then makes an equivalence query. This query asks if $M$ and $\hat{M}$ are equivalent. The counterexample from this equivalence query allows the algorithm to use membership queries to discover a new state of $M$. 

Angluin’s Learning Algorithm for DFA’s
Let $M$ be the target automaton, and assume $M$ is minimal. $\text{size}(M)$ is the number of states in $M$.

The learner attempts to continually discover new states of $M$. A new state is a state exhibiting behavior that is demonstrably different from the states discovered so far.

The learner runs in phases. In each phase, the learner constructs a hypothesis automaton $\hat{M}$ with states the currently discovered states of $M$.

The learning algorithm then makes an equivalence query. This query asks if $M$ and $\hat{M}$ are equivalent. The counterexample from this equivalence query allows the algorithm to use membership queries to discover a new state of $M$.

When all the states of $M$ have been discovered, $\hat{M} = M$. 

Angluin’s Learning Algorithm for DFA’s
Access Strings and Distinguishing Strings

How can the learning algorithm discover information about the states of $M$?

- Access Strings: Each string $s \in S$, when executed from the start state of $M$, leads to a unique state of $M$ that we denote by $M[s]$.
- Distinguishing Strings: For $s, s' \in S$ such that $s \neq s'$, there is a distinguishing string $d \in D$ such that one of $sd$ or $s'd$ reaches an accepting state of $M$, and the other reaches a rejecting state of $M$. 

Angluin’s Learning Algorithm for DFA’s
Access Strings and Distinguishing Strings

- How can the learning algorithm discover information about the states of $M$?
- The learner maintains a set $S$ consisting of at most $\text{size}(M)$ state access strings, and a set $D$ of distinguishing strings.

Angluin’s Learning Algorithm for DFA’s
Access Strings and Distinguishing Strings

How can the learning algorithm discover information about the states of $M$?

The learner maintains a set $S$ consisting of at most $\text{size}(M)$ state access strings, and a set $D$ of distinguishing strings.

- **Access Strings**: Each string $s \in S$, when executed from the start state of $M$, leads to a unique state of $M$ that we denote by $M[s]$.

- **Distinguishing Strings**: For $s, s' \in S$ such that $s \neq s'$, there is a distinguishing string $d \in D$ such that one of $sd$ or $s'd$ reaches an accepting state of $M$, and the other reaches a rejecting state of $M$. 
Access Strings and Distinguishing Strings

▶ How can the learning algorithm discover information about the states of $M$?

▶ The learner maintains a set $S$ consisting of at most $\text{size}(M)$ state access strings, and a set $D$ of distinguishing strings.
  
  ▶ Access Strings: Each string $s \in S$, when executed from the start state of $M$, leads to a unique state of $M$ that we denote by $M[s]$.
  
  ▶ Distinguishing Strings: For $s, s' \in S$ such that $s \neq s'$, there is a distinguishing string $d \in D$ such that one of $sd$ or $s'd$ reaches an accepting state of $M$, and the other reaches a rejecting state of $M$. 

Angluin’s Learning Algorithm for DFA’s
We shall refer to the states \( \{M[s] : s \in S\} \) as the \textit{known} states of \( M \), since we know how to access them from the start state.

Notice that all these known states must be distinct. This is because for \( s, s' \in S \), there is \( d \in D \) that witnesses the fact that starting from states \( M[s] \) and \( M[s'] \) and executing \( d \) leads to an accept state and to a reject state respectively, or vice versa.

The goal of the learning algorithm is to:

- discover the states of \( M \) by finding size(\( M \)) access strings;
- find distinguishing strings for each pair of strings in \( S \).

**Angluin’s Learning Algorithm for DFA’s**
We shall refer to the states \( \{ M[s] : s \in S \} \) as the *known* states of \( M \), since we know how to access them from the start state.

Notice that all these known states must be distinct. This is because for \( s, s' \in S \), there is \( d \in D \) that witnesses the fact that starting from states \( M[s] \) and \( M[s'] \) and executing \( d \) leads to an accept state and to a reject state respectively, or vice versa.
We shall refer to the states \( \{ M[s] : s \in S \} \) as the *known* states of \( M \), since we know how to access them from the start state.

Notice that all these known states must be distinct. This is because for \( s, s' \in S \), there is \( d \in D \) that witnesses the fact that starting from states \( M[s] \) and \( M[s'] \) and executing \( d \) leads to an accept state and to a reject state respectively, or vice versa.

The goal of the learning algorithm is to:
We shall refer to the states \( \{ M[s] : s \in S \} \) as the *known* states of \( M \), since we know how to access them from the start state.

Notice that all these known states must be distinct. This is because for \( s, s' \in S \), there is \( d \in D \) that witnesses the fact that starting from states \( M[s] \) and \( M[s'] \) and executing \( d \) leads to an accept state and to a reject state respectively, or vice versa.

The goal of the learning algorithm is to: discover the states of \( M \) by finding \( \text{size}(M) \) access strings;
We shall refer to the states \( \{ M[s] : s \in S \} \) as the *known* states of \( M \), since we know how to access them from the start state.

Notice that all these known states must be distinct. This is because for \( s, s' \in S \), there is \( d \in D \) that witnesses the fact that starting from states \( M[s] \) and \( M[s'] \) and executing \( d \) leads to an accept state and to a reject state respectively, or vice versa.

The goal of the learning algorithm is to:
- discover the states of \( M \) by finding \( \text{size}(M) \) access strings;
- to find distinguishing strings for each pair of strings in \( S \).
The sets $S$ and $D$ of access and distinguishing strings is maintained in a **binary classification tree**.
The sets $S$ and $D$ of access and distinguishing strings is maintained in a **binary classification tree**.

Each internal node of the binary classification tree is labeled by a string in $D$, and each leaf is labeled by a string in $S$. 

---

**Access Strings and Distinguishing Strings continue**

---

**Angluin’s Learning Algorithm for DFA's**
The sets $S$ and $D$ of access and distinguishing strings is maintained in a **binary classification tree**.

- Each internal node of the binary classification tree is labeled by a string in $D$, and each leaf is labeled by a string in $S$.
- The tree is constructed by placing at the root of a subtree of the classification tree a string $d$ from $D$, $s$, $s'$ ∈ $S$ are distinguished by the string labeling their least common ancestor in the classification tree.
The sets $S$ and $D$ of access and distinguishing strings is maintained in a **binary classification tree**.

Each internal node of the binary classification tree is labeled by a string in $D$, and each leaf is labeled by a string in $S$.

The tree is constructed by placing at the root of a subtree of the classification tree a string $d$ from $D$, and in the left subtree all strings $s \in S$ such that $sd$ is rejected by $M$, and in the right subtree all $s \in S$ such that $sd$ is accepted by $M$. 

Distinguishing strings $s, s' \in S$ are distinguished by the string labeling their least common ancestor in the classification tree.
The sets $S$ and $D$ of access and distinguishing strings is maintained in a **binary classification tree**.

Each internal node of the binary classification tree is labeled by a string in $D$, and each leaf is labeled by a string in $S$.

The tree is constructed by placing at the root of a subtree of the classification tree a string $d$ from $D$, and in the left subtree all strings $s \in S$ such that $sd$ is rejected by $M$, and in the right subtree all $s \in S$ such that $sd$ is accepted by $M$. 

Angluin’s Learning Algorithm for DFA’s
Access Strings and Distinguishing Strings continue

- The sets $S$ and $D$ of access and distinguishing strings is maintained in a **binary classification tree**.
- Each internal node of the binary classification tree is labeled by a string in $D$, and each leaf is labeled by a string in $S$.
- The tree is constructed by placing at the root of a subtree of the classification tree a string $d$ from $D$, and in the left subtree all strings $s \in S$ such that $sd$ is rejected by $M$, and in the right subtree all $s \in S$ such that $sd$ is accepted by $M$.
- $s, s' \in S$ are distinguished by the string labeling their least common ancestor in the classification tree.
The distinguishing string that labels the root of the classification tree is $\lambda$. This ensures that all the access strings to accepting states of $M$ will lie in the right subtree and the access strings to rejecting states in the left subtree. $\lambda$ must also be an access string. This ensures that we can access the start state of the automaton $M$. 

Angluin’s Learning Algorithm for DFA’s
The distinguishing string that labels the root of the classification tree is $\lambda$. This ensures that all the access strings to accepting states of $M$ will lie in the right subtree and the access strings to rejecting states in the left subtree.
The distinguishing string that labels the root of the classification tree is $\lambda$.
This ensures that all the access strings to accepting states of $M$ will lie in the right subtree and the access strings to rejecting states in the left subtree.

$\lambda$ must also be an access string.
The distinguishing string that labels the root of the classification tree is $\lambda$. This ensures that all the access strings to accepting states of $M$ will lie in the right subtree and the access strings to rejecting states in the left subtree.

$\lambda$ must also be an access string. This ensures that we can access the start state of the automaton $M$. 
Figure (a) shows an automaton that we will use as a running example. It accepts strings such that the number of 1’s is 3 modulo 4.
Figure (a) shows an automaton that we will use as a running example. It accepts strings such that the number of 1’s is 3 modulo 4.

Figure (b) shows a classification tree for this automaton, with access strings \( \{ \lambda, 110, 1101 \} \) and distinguishing strings \( \{ \lambda, 1 \} \).
Computing a state partition

- Suppose $s'$ is not in the current access string set $S$, but that $M[s'] = M[s]$ for some $s \in S$. 

...
Computing a state partition

- Suppose $s'$ is not in the current access string set $S$, but that $M[s'] = M[s]$ for some $s \in S$.
- We determine $s$ from $s'$ by sifting $s'$ down our classification tree using membership queries.

Angluin’s Learning Algorithm for DFA’s
Computing a state partition

- Suppose $s'$ is not in the current access string set $S$, but that $M[s'] = M[s]$ for some $s \in S$.
- We determine $s$ from $s'$ by **sifting** $s'$ down our classification tree using membership queries.
- Starting at the root, if we are at an internal node labeled by the distinguishing string $d$, we make a membership query on the string $s'd$ and go to the left or right subtree as indicated by the query answer (left on reject, right on accept).
Computing a state partition

- Suppose $s'$ is not in the current access string set $S$, but that $M[s'] = M[s]$ for some $s \in S$.
- We determine $s$ from $s'$ by **sifting** $s'$ down our classification tree using membership queries.
- Starting at the root, if we are at an internal node labeled by the distinguishing string $d$, we make a membership query on the string $s'd$ and go to the left or right subtree as indicated by the query answer (left on reject, right on accept).
- Even if $M[s'] \neq M[s]$ for all $s \in S$, sifting $s'$ down the classification tree still defines a path to a leaf depending only on $M[s']$. 

Angluin’s Learning Algorithm for DFA’s
Computing a state partition

- Suppose $s'$ is not in the current access string set $S$, but that $M[s'] = M[s]$ for some $s \in S$.
- We determine $s$ from $s'$ by sifting $s'$ down our classification tree using membership queries.
- Starting at the root, if we are at an internal node labeled by the distinguishing string $d$, we make a membership query on the string $s'd$ and go to the left or right subtree as indicated by the query answer (left on reject, right on accept).
- Even if $M[s'] \neq M[s]$ for all $s \in S$, sifting $s'$ down the classification tree still defines a path to a leaf depending only on $M[s']$.
- If $M[s'] = M[s'']$ then sifting $s'$ and $s''$ defines exactly the same path in the classification tree.
Computing a state partition

- Suppose \( s' \) is not in the current access string set \( S \), but that \( M[s'] = M[s] \) for some \( s \in S \).
- We determine \( s \) from \( s' \) by **sifting** \( s' \) down our classification tree using membership queries.
- Starting at the root, if we are at an internal node labeled by the distinguishing string \( d \), we make a membership query on the string \( s'd \) and go to the left or right subtree as indicated by the query answer (left on reject, right on accept).
- Even if \( M[s'] \neq M[s] \) for all \( s \in S \), sifting \( s' \) down the classification tree still defines a path to a leaf depending only on \( M[s'] \).
- If \( M[s'] = M[s'''] \) then sifting \( s' \) and \( s''' \) defines exactly the same path in the classification tree.
- The classification tree partitions the states of \( M \). Each equivalence class contains exactly one state \( M[s] \) such that \( s \in S \).
This Figure shows the partition of the previously given automaton and classification tree. The known state in each equivalence class is shaded.
We identify (label) the states of $M$ and $\hat{M}$ with the access strings in the classification tree.
We identify (label) the states of $M$ and $\hat{M}$ with the access strings in the classification tree.

For each access string (state) $s$ and symbol $b$, the destination state of the $b$-transition out of state $s$ is just the access string that results from sifting $sb$ down the classification tree.
(a) The transitions of $M$ are represented by dashed lines, and the states are grouped by the equivalence classes defined by the classification tree. $M$ is a four-state automaton, with transitions represented by dashed lines. The states of $M$ are partitioned into two classes of two states each. Each known state $M[s]$ for $s \in S$ is shaded.
(a) The transitions of $M$ are represented by dashed lines, and the states are grouped by the equivalence classes defined by the classification tree. $M$ is a four-state automaton, with transitions represented by dashed lines. The states of $M$ are partitioned into two classes of two states each. Each known state $M[s]$ for $s \in S$ is shaded.

(b) $\hat{M}$ extracted. $\hat{M}$ is defined only on shaded states, and each equivalence class of $M$ has exactly one such shaded state.
The hypothesis $\hat{M}$ continue

(a) The transitions of $M$ are represented by dashed lines, and the states are grouped by the equivalence classes defined by the classification tree. $M$ is a four-state automaton, with transitions represented by dashed lines. The states of $M$ are partitioned into two classes of two states each. Each known state $M[s]$ for $s \in S$ is shaded.

(b) $\hat{M}$ extracted. $\hat{M}$ is defined only on shaded states, and each equivalence class of $M$ has exactly one such shaded state. The transitions of $\hat{M}$, represented by bold lines, are defined by taking a transition leaving a shaded state and redirecting the target to the unique shaded state of the equivalence class of the destination state.
Next we show how to use a counterexample $\gamma$ to discover a new state of $M$. 

The idea is to simulate the behavior of both $M$ and $\hat{M}$ in parallel on $\gamma$ to discover the first point at which the two trajectories diverge to different equivalence classes of states. At the point of divergence, the dashed and bold transitions must take place from two different states in the same equivalence class, thus providing us with access to a new state.
Next we show how to use a counterexample $\gamma$ to discover a new state of $M$.

The idea is to simulate the behavior of both $M$ and $\hat{M}$ in parallel on $\gamma$ to discover the first point at which the two trajectories diverge to different equivalence classes of states.
Next we show how to use a counterexample $\gamma$ to discover a new state of $M$.

The idea is to simulate the behavior of both $M$ and $\hat{M}$ in parallel on $\gamma$ to discover the first point at which the two trajectories diverge to different equivalence classes of states.

At the point of divergence, the dashed and bold transitions must take place from two different states in the same equivalence class, thus providing us with access to a new state.
Recall that the root of the classification tree is labeled by $\lambda$. This implies that no equivalence class of $M$ contains both an accepting and a rejecting state.
Recall that the root of the classification tree is labeled by $\lambda$. This implies that no equivalence class of $M$ contains both an accepting and a rejecting state.

Also, since one of the access strings is $\lambda$, $M$ and $\hat{M}$ are synchronized at the start of processing a string.
Recall that the root of the classification tree is labeled by $\lambda$. This implies that no equivalence class of $M$ contains both an accepting and a rejecting state.

Also, since one of the access strings is $\lambda$, $M$ and $\hat{M}$ are synchronized at the start of processing a string.

So the dashed and bold trajectories determined by the counterexample $\gamma$ begin in a common equivalence class (in fact, in the same state) and end up in different equivalence classes (since exactly one of $M$ and $\hat{M}$ accepts $\gamma$).
\( M[s] \) and \( \hat{M}[s] \) denote the states reached by following the transitions of \( M \) and \( \hat{M} \) respectively on \( s \).
Using a Counterexample continue

- $M[s]$ and $\widehat{M}[s]$ denote the states reached by following the transitions of $M$ and $\widehat{M}$ respectively on $s$.

- $\gamma_i$ denotes the $i$th symbol of $\gamma$ and $\gamma[i] = \gamma_1 \ldots \gamma_i$. 
Using a Counterexample continue

- \( M[s] \) and \( \hat{M}[s] \) denote the states reached by following the transitions of \( M \) and \( \hat{M} \) respectively on \( s \).
- \( \gamma_i \) denotes the \( i \)th symbol of \( \gamma \) and \( \gamma[i] = \gamma_1 \ldots \gamma_i \).
- \( j \) is the smallest index such that \( M[\gamma[j]] \neq \hat{M}[\gamma[j]] \).

Angluin’s Learning Algorithm for DFA’s
By the choice of $j$, $M[\gamma[j - 1]]$ and $\hat{M}[\gamma[j - 1]]$ are in the same equivalence class, but the dashed transition from $M[\gamma[j - 1]]$ and the bold transition from $\hat{M}[\gamma[j - 1]]$ on $\gamma_j$ leads to different equivalence classes.
Using a Counterexample continue

By the choice of \( j \), \( M[\gamma[j - 1]] \) and \( \hat{M}[\gamma[j - 1]] \) are in the same equivalence class, but the dashed transition from \( M[\gamma[j - 1]] \) and the bold transition from \( \hat{M}[\gamma[j - 1]] \) on \( \gamma_j \) leads to different equivalence classes.

Thus \( M[\gamma[j - 1]] \) and \( \hat{M}[\gamma[j - 1]] \) are different states in the same equivalence class.
By the choice of \( j \), \( M[\gamma[j - 1]] \) and \( \hat{M}[\gamma[j - 1]] \) are in the same equivalence class, but the dashed transition from \( M[\gamma[j - 1]] \) and the bold transition from \( \hat{M}[\gamma[j - 1]] \) on \( \gamma_j \) leads to different equivalence classes.

Thus \( M[\gamma[j - 1]] \) and \( \hat{M}[\gamma[j - 1]] \) are different states in the same equivalence class.

If \( d \) is the string distinguishing the equivalence classes of \( M[\gamma[j]] \) and \( \hat{M}[\gamma[j]] \), then \( \gamma_j d \) distinguishes \( M[\gamma[j - 1]] \) and \( \hat{M}[\gamma[j - 1]] \).
While the number of leaves of the classification tree is smaller than \( \text{size}(M) \), \( \hat{M} \neq M \).
While the number of leaves of the classification tree is smaller than \( \text{size}(M) \), \( \hat{M} \neq M \).

An equivalence query returns a counterexample \( \gamma \) that can be used to update the classification tree by adding a new leaf node.
Using a Counterexample continue

- While the number of leaves of the classification tree is smaller than \( \text{size}(M) \), \( \hat{M} \neq M \).
- An equivalence query returns a counterexample \( \gamma \) that can be used to update the classification tree by adding a new leaf node.
- Eventually the classification tree will have \( \text{size}(M) \) leaf nodes, each accessing a different state of \( M \), and at this stage \( M = \hat{M} \).
We start by describing the subroutine \textbf{Sift}. This subroutine takes as input a string \( s \) and the current classification tree \( T \), and outputs the access string in \( T \) of the equivalence class of \( M[s] \).
We start by describing the subroutine **Sift**. This subroutine takes as input a string $s$ and the current classification tree $T$, and outputs the access string in $T$ of the equivalence class of $M[s]$.

**Procedure Sift**($s, T$):
- Initialisation: set the current node to be the root node of $T$. 
We start by describing the subroutine \textbf{Sift}. This subroutine takes as input a string $s$ and the current classification tree $T$, and outputs the access string in $T$ of the equivalence class of $M[s]$.

\textbf{Procedure Sift}($s, T$):

- **Initialisation:** set the current node to be the root node of $T$.
- **Main Loop:**
We start by describing the subroutine \textbf{Sift}. This subroutine takes as input a string $s$ and the current classification tree $T$, and outputs the access string in $T$ of the equivalence class of $M[s]$.

\textbf{Procedure Sift}$(s, T)$:

- Initialisation: set the current node to be the root node of $T$.
- Main Loop:
  - Let $d$ be the distinguishing string at the current node in the tree.
We start by describing the subroutine **Sift**. This subroutine takes as input a string $s$ and the current classification tree $T$, and outputs the access string in $T$ of the equivalence class of $M[s]$.

**Procedure Sift**$(s, T)$:

- **Initialisation**: set the current node to be the root node of $T$.
- **Main Loop**:
  - Let $d$ be the distinguishing string at the current node in the tree.
  - Make a membership query on $sd$. If $sd$ is accepted by $M$, update the current node to be the right child of the current node. Otherwise, update the current node to be the left child of the current node.
We start by describing the subroutine \textbf{Sift}. This subroutine takes as input a string $s$ and the current classification tree $T$, and outputs the access string in $T$ of the equivalence class of $M[s]$.

\textbf{Procedure Sift}(s, T):

- Initialisation: set the current node to be the root node of $T$.

- Main Loop:
  - Let $d$ be the distinguishing string at the current node in the tree.
  - Make a membership query on $sd$. If $sd$ is accepted by $M$, update the current node to be the right child of the current node. Otherwise, update the current node to be the left child of the current node.
  - If the current node is a leaf node, then return the access string stored at this leaf. Otherwise, repeat the Main Loop.
Algorithm for learning DFA’s - Procedure

Tentative-Hypothesis($T$)

Next, we describe the procedure for constructing the hypothesis automaton $\hat{M}$ that is defined by the current classification tree $T$. 
Next, we describe the procedure for constructing the hypothesis automaton $\hat{M}$ that is defined by the current classification tree $T$.

**Procedure Tentative-Hypothesis($T$):**

- For each access string (leaf) of $T$, create a state in $\hat{M}$ that is labeled by that access string. The start state of $\hat{M}$ is $\lambda$. 

Angluin’s Learning Algorithm for DFA’s
Algorithm for learning DFA’s - Procedure
Tentative-Hypothesis(\(T\))

Next, we describe the procedure for constructing the hypothesis automaton \(\hat{M}\) that is defined by the current classification tree \(T\).

**Procedure Tentative-Hypothesis(\(T\)):**

- For each access string (leaf) of \(T\), create a state in \(\hat{M}\) that is labeled by that access string. The start state of \(\hat{M}\) is \(\lambda\).
- For each access state \(s\) of \(\hat{M}\) and each alphabet symbol \(b\), compute the \(b\)-transition out of state \(s\) in \(\hat{M}\) as follows:
  - \(s' \leftarrow \text{Sift}(sb, T)\).
Next, we describe the procedure for constructing the hypothesis automaton $\hat{M}$ that is defined by the current classification tree $T$.

**Procedure Tentative-Hypothesis($T$):**

- For each access string (leaf) of $T$, create a state in $\hat{M}$ that is labeled by that access string. The start state of $\hat{M}$ is $\lambda$.
- For each access state $s$ of $\hat{M}$ and each alphabet symbol $b$, compute the $b$-transition out of state $s$ in $\hat{M}$ as follows:
  - $s' \leftarrow \text{Sift}(sb, T)$.
  - Direct the $b$-transition out of state $s$ to state $s'$. 

**Angluin’s Learning Algorithm for DFA’s**
Next, we describe the procedure for constructing the hypothesis automaton $\hat{M}$ that is defined by the current classification tree $T$.

**Procedure Tentative-Hypothesis($T$):**

- For each access string (leaf) of $T$, create a state in $\hat{M}$ that is labeled by that access string. The start state of $\hat{M}$ is $\lambda$.
- For each access state $s$ of $\hat{M}$ and each alphabet symbol $b$, compute the $b$-transition out of state $s$ in $\hat{M}$ as follows:
  - $s' \leftarrow \text{Sift}(sb, T)$.
  - Direct the $b$-transition out of state $s$ to state $s'$.
- Return $\hat{M}$
Next we describe the procedure **Update-Tree**, which takes as arguments the current classification tree $T$ and a counterexample string $\gamma$ to the hypothesis automaton $\hat{M}$ defined by $T$. 
Next we describe the procedure \textbf{Update-Tree}, which takes as arguments the current classification tree $T$ and a counterexample string $\gamma$ to the hypothesis automaton $\hat{M}$ defined by $T$.

The procedure finds a new access string, and updates $T$ by adding a new leaf node labeled with the new access string.
Algorithm for learning DFA’s - Procedure Update-Tree

Procedure Update-Tree($\gamma, T$):

For each prefix $\gamma[i]$ of $\gamma$:
- $s_i \leftarrow \text{Sift}(\gamma[i], T)$.
- Let $\hat{s}_i = \hat{M}[\gamma[i]]$.
- Let $j$ be as small as possible such that $s_j \neq \hat{s}_j$.
- Replace the node labeled with the access string $s_{j-1}$ in $T$ with an internal node with two leaf nodes. One leaf node is labeled with the access string $s_{j-1}$ and the other with the new access string $\gamma[j-1]$. The newly created internal node is labeled with the distinguishing string $\gamma_jd$, where $d$ is the distinguishing string for $s_j$ and $\hat{s}_j$.

Angluin’s Learning Algorithm for DFA’s
Algorithm for learning DFA’s - Procedure Update-Tree

Procedure Update-Tree(γ, T):

▶ For each prefix γ[i] of γ:

s_i ← Sift(γ[i], T).

Let ˆs_i = ˆM[γ[i]].

Let j be as small as possible such that s_j ≠ ˆs_j.

Replace the node labeled with the access string s_j−1 in T with an internal node with two leaf nodes. One leaf node is labeled with the access string s_j−1 and the other with the new access string γ[j−1]. The newly created internal node is labeled with the distinguishing string γ_j, where d is the distinguishing string for s_j and ˆs_j.
Algorithm for learning DFA's - Procedure Update-Tree

**Procedure Update-Tree**($\gamma, T$):

- For each prefix $\gamma[i]$ of $\gamma$:
  - $s_i \leftarrow \text{Sift}(\gamma[i], T)$.

Angluin's Learning Algorithm for DFA's
Procedure Update-Tree($\gamma, T$):

- For each prefix $\gamma[i]$ of $\gamma$:
  - $s_i \leftarrow \text{Sift}(\gamma[i], T)$.
  - Let $\hat{s}_i = \hat{M}[\gamma[i]]$.
Procedure Update-Tree($\gamma$, $T$):

- For each prefix $\gamma[i]$ of $\gamma$:
  - $s_i \leftarrow \text{Sift}(\gamma[i], T)$.
  - Let $\hat{s}_i = \hat{M}[\gamma[i]]$.
- Let $j$ be as small as possible such that $s_j \neq \hat{s}_j$. 

Angluin's Learning Algorithm for DFA’s
Algorithm for learning DFA’s - Procedure Update-Tree

Procedure Update-Tree($\gamma, T$):

- For each prefix $\gamma[i]$ of $\gamma$:
  - $s_i \leftarrow \text{Sift}(\gamma[i], T)$.
  - Let $\hat{s}_i = \hat{M}[\gamma[i]]$.

- Let $j$ be as small as possible such $s_j \neq \hat{s}_j$.

- Replace the node labeled with the access string $s_{j-1}$ in $T$ with an internal node with two leaf nodes.

Angluin’s Learning Algorithm for DFA’s
Procedure Update-Tree($\gamma, T$):

- For each prefix $\gamma[i]$ of $\gamma$:
  - $s_i \leftarrow \text{Sift}(\gamma[i], T)$.
  - Let $\hat{s}_i = \hat{M}[\gamma[i]]$.

- Let $j$ be as small as possible such that $s_j \neq \hat{s}_j$.

- Replace the node labeled with the access string $s_{j-1}$ in $T$ with an internal node with two leaf nodes. One leaf node is labeled with the access string $s_{j-1}$ and the other with the new access string $\gamma[j-1]$. 
Algorithm for learning DFA’s - Procedure Update-Tree

Procedure Update-Tree(γ, T):

- For each prefix γ[i] of γ:
  - $s_i \leftarrow \text{Sift}(γ[i], T)$.
  - Let $\hat{s}_i = \hat{M}[γ[i]]$.

- Let $j$ be as small as possible such $s_j \neq \hat{s}_j$.

- Replace the node labeled with the access string $s_{j-1}$ in $T$ with an internal node with two leaf nodes. One leaf node is labeled with the access string $s_{j-1}$ and the other with the new access string $γ[j − 1]$. The newly created internal node is labeled with the distinguishing string $γ_j \cdot d$, where $d$ is the distinguishing string for $s_j$ and $\hat{s}_j$. 
Algorithm for learning DFA’s - Overall Algorithm

- Initialization:
  - Do a membership query on the string \( \lambda \) to determine whether the start state of \( M \) is accepting or rejecting.
  - Construct a hypothesis automaton that consists simply of this single (accepting or rejecting) state with selfloops for both the all transitions.
  - Perform an equivalence query on this automaton; let the counterexample string be \( \gamma \).
  - Initialize the classification tree \( T \) to have a root labeled with the distinguishing string \( \lambda \) and two leaves labeled with access strings \( \lambda \) and \( \gamma \).
Algorithm for learning DFA’s - Overall Algorithm

- Initialization:
  - Do a membership query on the string $\lambda$ to determine whether the start state of $M$ is accepting or rejecting.
Algorithm for learning DFA’s - Overall Algorithm

- Initialization:
  - Do a membership query on the string $\lambda$ to determine whether the start state of $M$ is accepting or rejecting.
  - Construct a hypothesis automaton that consists simply of this single (accepting or rejecting) state with selfloops for both the all transitions.
Initialization:

- Do a membership query on the string $\lambda$ to determine whether the start state of $M$ is accepting or rejecting.
- Construct a hypothesis automaton that consists simply of this single (accepting or rejecting) state with selfloops for both the all transitions.
- Perform an equivalence query on this automaton; let the counterexample string be $\gamma$. 

Angluin’s Learning Algorithm for DFA’s
Algorithm for learning DFA’s - Overall Algorithm

Initialization:
- Do a membership query on the string $\lambda$ to determine whether the start state of $M$ is accepting or rejecting.
- Construct a hypothesis automaton that consists simply of this single (accepting or rejecting) state with selfloops for both the all transitions.
- Perform an equivalence query on this automaton; let the counterexample string be $\gamma$.
- Initialize the classification tree $T$ to have a root labeled with the distinguishing string $\lambda$ and two leaves labeled with access strings $\lambda$ and $\gamma$. 
Main Loop:

- Let $T$ be the current classification tree.
- $\hat{M} \leftarrow \text{Tentative-Hypothesis}(T)$.
- Make an equivalence query on $\hat{M}$.
  - If it is equivalent to the target then output $\hat{M}$ and halt.
  - Otherwise, let $\gamma$ be the counterexample string.
- $\text{Update-Tree}(T, \gamma)$.
- Repeat Main Loop.

Note that the learning algorithm can reuse a specific counterexample $\gamma$ until $\hat{M}$ accepts $\gamma$ if $M$ accepts, or rejects $\gamma$ if $M$ does.
Main Loop:
- Let $T$ be the current classification tree.

Note that the learning algorithm can reuse a specific counterexample $\gamma$ until $\hat{M}$ accepts $\gamma$ if $M$ does, or rejects $\gamma$ if $M$ does.
Algorithm for learning DFA’s - Overall Algorithm continue

- **Main Loop:**
  - Let $T$ be the current classification tree.
  - $\hat{M} \leftarrow \text{Tentative-Hypothesis}(T)$.
Main Loop:
- Let $T$ be the current classification tree.
- $\hat{M} \leftarrow \text{Tentative-Hypothesis}(T)$.
- Make an equivalence query on $\hat{M}$. If it is equivalent to the target then output $\hat{M}$ and halt. Otherwise, let $\gamma$ be the counterexample string.
- Update-Tree($T, \gamma$).
- Repeat Main Loop.

Note that the learning algorithm can reuse a specific counterexample $\gamma$ until $\hat{M}$ accepts $\gamma$ if $M$ does, or rejects $\gamma$ if $M$ does.
Algorithm for learning DFA’s - Overall Algorithm continue

Main Loop:
- Let $T$ be the current classification tree.
- $\hat{M} \leftarrow$ Tentative-Hypothesis($T$).
- Make an equivalence query on $\hat{M}$. If it is equivalent to the target then output $\hat{M}$ and halt.
Algorithm for learning DFA’s - Overall Algorithm continue

Main Loop:
- Let $T$ be the current classification tree.
- $\hat{M} \leftarrow$ Tentative-Hypothesis($T$).
- Make an equivalence query on $\hat{M}$. If it is equivalent to the target then output $\hat{M}$ and halt. Otherwise, let $\gamma$ be the counterexample string.

Note that the learning algorithm can reuse a specific counterexample $\gamma$ until $\hat{M}$ accepts $\gamma$ if $M$ does, or rejects $\gamma$ if $M$ does.
Main Loop:
- Let $T$ be the current classification tree.
- $\hat{M} \leftarrow \text{Tentative-Hypothesis}(T)$.
- Make an equivalence query on $\hat{M}$. If it is equivalent to the target then output $\hat{M}$ and halt. Otherwise, let $\gamma$ be the counterexample string.
- $\text{Update-Tree}(T, \gamma)$. 

Angluin’s Learning Algorithm for DFA’s
Algorithm for learning DFA’s - Overall Algorithm continue

Main Loop:
- Let $T$ be the current classification tree.
- $\hat{M} \leftarrow$ Tentative-Hypothesis($T$).
- Make an equivalence query on $\hat{M}$. If it is equivalent to the target then output $\hat{M}$ and halt. Otherwise, let $\gamma$ be the counterexample string.
- Update-Tree($T, \gamma$).
- Repeat Main Loop.

Note that the learning algorithm can reuse a specific counterexample $\gamma$ until $\hat{M}$ accepts $\gamma$ if $M$ does, or rejects $\gamma$ if $M$ does.
The Figure on the next slide shows the execution of the learning algorithm on the 3 mod 4 counter automaton.
The Figure on the next slide shows the execution of the learning algorithm on the $3 \mod 4$ counter automaton.

The first column shows the target automaton with the partition defined by the classification tree of the previous row, along with the shaded known states.
The Figure on the next slide shows the execution of the learning algorithm on the $3 \mod 4$ counter automaton.

The first column shows the target automaton with the partition defined by the classification tree of the previous row, along with the shaded known states.

The second column shows the hypothesis $\hat{M}$ defined by the partition to its left.

The third column shows the prefix $\gamma[j-1]$ of $\gamma = 1011$, with $j$ as small as possible such that $M[\gamma[j]]$ and $\hat{M}[\gamma[j]]$ are not in the partition.

The fourth column shows the classification tree at each step.
Example

- The Figure on the next slide shows the execution of the learning algorithm on the $3 \mod 4$ counter automaton.
- The first column shows the target automaton with the partition defined by the classification tree of the previous row, along with the shaded known states.
- The second column shows the hypothesis $\hat{M}$ defined by the partition to its left.
- Every equivalence query on $\hat{M}$ is answered by the same repeated counterexample 1101, which is accepted by $M$, until $\hat{M} = M$. In most cases more than one counterexample is required.
Example

- The Figure on the next slide shows the execution of the learning algorithm on the 3 mod 4 counter automaton.
- The first column shows the target automaton with the partition defined by the classification tree of the previous row, along with the shaded known states.
- The second column shows the hypothesis $\hat{M}$ defined by the partition to its left.
- Every equivalence query on $\hat{M}$ is answered by the same repeated counterexample 1101, which is accepted by $M$, until $\hat{M} = M$. In most cases more than one counterexample is required.
- The third column shows the prefix $\gamma[j - 1]$ of $\gamma = 1011$, with $j$ as small as possible such that $M[\gamma[j]]$ and $\hat{M}[\gamma[j]]$ are not in the partition.
The Figure on the next slide shows the execution of the learning algorithm on the 3 \mod 4 counter automaton.

The first column shows the target automaton with the partition defined by the classification tree of the previous row, along with the shaded known states.

The second column shows the hypothesis $\hat{M}$ defined by the partition to its left.

Every equivalence query on $\hat{M}$ is answered by the same repeated counterexample 1101, which is accepted by $M$, until $\hat{M} = M$. In most cases more than one counterexample is required.

The third column shows the prefix $\gamma[j - 1]$ of $\gamma = 1011$, with $j$ as small as possible such that $M[\gamma[j]]$ and $\hat{M}[\gamma[j]]$ are not in the partition.

The fourth column shows the classification tree at each step.
Angluin’s Learning Algorithm for DFA’s
Homework 4

You may assume that the input alphabet $\Sigma$ of your DFA’s are $\{0, 1\}$ or equivalently $\{a, b\}$.

You should implement in Java Angluin’s algorithm as explained in these slides. A report on your implementation of at least 5 pages should also be written. In the report you should describe how your implementation learns at least 3 different minimal automata (as in the Figure on the previous slide), where each minimal DFA has at least 4 states. You should attempt to make the implementation as efficient as possible and try to minimise the number of equivalence and membership queries by for example storing the answers to previous membership queries.

Your implementation should have the following components:

▶ a random automaton generator: The user should specify a number of states $n$ and a number of accept states $a$ ($a \leq n$) for an automaton. The random automaton generator should randomly generate a DFA $M'$ with $n$ states, with all $n$ states reachable from the initial state, and with $a$ accept states.

▶ a DFA minimiser: The minimiser should minimise a DFA $M'$, received for example from the random DFA generator, to an equivalent minimal DFA $M$.

▶ an Angluin teacher: The constructor of the Angluin teacher should take a random minimal DFA $M$ as parameter. Given a string $s$ as input, the teacher should say if $M$ accepts $s$ or not. Given an automaton $\hat{M}$, the teacher should say if $M = \hat{M}$, or should return a (random) counterexample.

▶ an Angluin learner: The learner should construct an automaton $\hat{M}$ by asking the teacher the membership queries and should update $\hat{M}$ when receiving a counterexample from the teacher.