1 Finite automata

A **finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states
- Σ is the finite alphabet
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accept states (final states)

A finite automaton reads an input cell-by-cell, transitioning between states depending on the current input symbol and state.

If A is the set of all strings accepted by a machine M, A is the **language** of M, denoted $L(M) = A$. If A is the language of M, M **recognises** A.

Can also say that M accepts A, but this is confusing – say "accept" when referring to machines accepting strings, and "recognise" when referring to machines accepting languages.

A machine only recognises one language, but can accept several strings. If a machine accepts no strings, it recognises the empty language $\emptyset$.

Question. Is zero accept states allowable?
Answer. Yes. Set F to be the empty set $\emptyset$, yielding zero accept states.

Question. Must there be exactly one transition exiting every state for each possible input symbol?
Answer. Yes. The transition function, δ, specifies one successor state for each possible combination of a state and an input symbol.

Finite automata are good models for computers with extremely limited memory.

1.1 Computation

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and $w = w_1 w_2 ... w_n$ be a string, where each w_i is a member of alphabet Σ. M accepts w is there exists a sequence of states $r_0, r_1, ..., r_n \in Q$ where:

1. $r_0 = q_0$ (machine starts in the start state),
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, ..., n-1$ (machine goes between states according to the transition function), and
3. $r_n \in F$ (machine accepts an input if it ends in an accept state).

The machine M recognises language A if $A = \{w | M \text{ accepts } w\}$

2 Nondeterminism

In deterministic computation, when a machine is in a given state and reads the next input symbol, its next state is known (determined). In nondeterministic machines, there can be several choices for the next state.

Nondeterminism is just a generalisation of determinism. Every deterministic finite automaton (DFA) is automatically a nondeterministic finite automaton (NDFA).

Every state of a DFA has one exiting transition arrow for each symbol in the alphabet. In NDFAs, a state can have zero, one, or many exiting arrows for each symbol in the alphabet.

In DFAs, labels on transition arrows must be symbols from the alphabet. In NDFAs, labels on transition arrows may be symbols from the alphabet, or ϵ. There can be zero, one, or many arrows exiting each state with label ϵ.

2.1 How do nondeterministic machines compute?

Suppose a NDFA is running on an input string and gets to a state with multiple ways to proceed. After reading the input symbol, the machine splits into multiple copies of itself, and follows all possibilities in parallel. Each new machine takes one of the possibilities and continues.

If ϵ-labelled exiting arrows are encountered, the machine splits into multiple copies – one for each of the ϵ-labelled arrows and one staying at the current state – without reading an input.

In any of the new machines, if the next input symbol does not appear on any arrows exiting the current state, the machine (and the branch of computation associated with it) dies.

If any one of these machines is in an accept state at the end of the input, the input string is accepted by the NDFA.

Summary. Nondeterministic computation can viewed as a tree of possibilities, with the root being the start of the computation, and every branching point (fork) corresponds to a point where the machine has multiple choices.

If one of the branches ends in an accept state, the machine accepts.

Nondeterminism can be considered parallel computation wherein multiple independent "processes" or "threads" can be running concurrently.

2.2 Nondeterministic finite automata

DFAs and NDFAs differ in the type of transition function:

- In DFAs, the transition function takes a state and an input symbol, and produces the next state

- In NDFAs, the transition function takes a state and an input symbol or the empty string, and produces the set of possible next states

A **nondeterministic finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states
- Σ is the finite alphabet
- $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accept states (final states)

$\mathcal{P}(Q)$ is the power set (set of all subsets) of a set Q. For any alphabet Σ, write Σ_ϵ to mean $\Sigma \cup \{\epsilon\}$.

The **formal definition of computation** for a NDFA is also similar for that of a DFA. Let $N = (Q, \Sigma, \delta, q_0, F)$ be a NDFA and w be a string over alphabet Σ. N accepts w if w can be written as $w = y_1 y_2 ... y_m$, where each y_i is a member of Σ_ϵ and there exists a sequence of states $r_0, r_1, ..., r_m \in Q$ where:

1. $r_0 = q_0$ (machine starts in the start state),
2. $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, ..., m-1$ (state r_{i+1} is one of the allowable next states when N is in state r_i and reading y_{i+1}), and
3. $r_m \in F$ (machine accepts an input if it ends in an accept state).

Note that $\delta(r_i, y_{i+1})$ is the set of allowable next states, so say that r_{i+1} is a member of this set.

2.3 How are NDFAs useful?

- Every NDFA can be converted into an equivalent DFA – it is sometimes easier to construct NDFAs than directly constructing DFAs
- NDFAs can be much smaller than their DFA counterparts
- The functioning of some NDFAs is easier to understand than that of their DFA counterparts
- "Nondeterminism in finite automata is a good introduction to nondeterminism in more powerful computational models"

2.4 Equivalence

Two machines are **equivalent** if they recognise the same language.

Theorem. Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

If k is the number of states of the NDFA, it has 2^k subsets of states. Each subset corresponds to one of the possibilities that the DFA must remember, so there will be 2^k states in the DFA simulating the NDFA.

Proof. Let $N = (Q, \Sigma, \delta, q_0, F)$ be the NDFA recognising some language A. Construct DFA $M = (Q', \Sigma, \delta', q_0', F')$ recognising A.

For any state R of M, define $E(R)$ to be the collection of states that can be reached from members of R only by going along ϵ arrows, including the members of R themselves. For $R \subseteq Q$ let:

$$E(R) = \{q | q \text{ can be reached from } R \text{ by travelling along 0 or more } \epsilon \text{ arrows}\}$$

The construction of M is as follows:

- $Q' = \mathcal{P}(Q)$ – every state of M is a set of states in N.
- For $R \in Q'$ and $a \in \Sigma$, let $\delta'(R, a) = \{q \in Q | q \in E(\delta(r, a))$ for some $r \in R\}$ – if R is a state of M, it is also a set of states in N. When M reads a symbol a in state R, it shows where a takes each state in R. As each state may go to a set of states, take the union of all these sets.
- $q_0' = E(\{q_0\})$ – M starts in the state corresponding to the set containing just the start state of N.
- $F' = \{R \in Q' | R$ contains an accept state of $N\}$ – M accepts if one of the possible states that N could be in at any point is an accept state. □

3 Regular languages

A language is a **regular language** if it is recognised by some finite automaton.

Theorem. A language is regular if and only if some nondeterministic finite automaton recognises it.

Proof.

($\rightarrow$)	As any NDFA can be converted into an equivalent DFA, if an NDFA recognises some languages, so does some DFA, and the language is regular.
($\leftarrow$)	A regular language has a DFA recognising it, and any DFA is also an NDFA.

3.1 Regular operations

Define three operations and language, called **regular operations**, to study properties of the regular languages.

Let A and B be languages.

- **Union:** $A \cup B = \{x | x \in A \text{ or } x \in B\}$
  Takes all strings in both A and B and puts them in one language.
- **Concatenation:** $A \circ B = \{xy | x \in A \text{ and } y \in B\}$
  Attaches a string from A in front of a string from B, in all possible ways, to get the strings in the new language.
- **Star:** $A^* = \{x_1 x_2 ... x_k | k \geq 0 \text{ and each } x_i \in A\}$
  A unary operation that attaches any number of strings in A together to get a string in the new language.
  As "any number" includes zero, ϵ is always a member of A^*.

3.2 Closure under operations

Theorem. The class of regular languages is closed under union.

Proof. Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognise A_1 and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognise A_2. Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognise $A_1 \cup A_2$.

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$
   The states of N are all the states of N_1 and N_2, with the addition of a new start state, q_0.
2. State q_0 is the start state of N
3. The set of accept states is $F = F_1 \cup F_2$
   N accepts if either N_1 or N_2 accepts, so the accept states of N are all those of N_1 and N_2.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\epsilon$:

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \epsilon \\ \emptyset & q = q_0 \text{ and } a \neq \epsilon \end{cases}$$

Theorem. The class of regular languages is closed under concatenation.

Proof. Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognise A_1 and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognise A_2. Construct $N = (Q, \Sigma, \delta, q_1, F_2)$ to recognise $A_1 \circ A_2$.

1. $Q = Q_1 \cup Q_2$
2. The state q_1 is the same as the start state of N_1

3. Accept states F_2 are the same as the accept states of N_2.
   The accept states of N_1 have additional ϵ arrows that nondeterministically allow branching to N_2 whenever N_1 is in an accept state, signifying that N_1 has found an initial piece of the input that constitutes a string in A_1.
   N accepts when the input can be split into two parts – the first accepted by N_1 and the second by N_2.
   N nondeterministically "guesses" where to make the split.

4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\epsilon$:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \epsilon \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1 \text{ and } a = \epsilon \\ \delta_2(q,a) & q \in Q_2 \end{cases}$$

Theorem. The class of regular languages is closed under star.

Given a regular language A_1, want to prove that A_1^* is also regular. Take a NDFA N_1 recognising A_1 and modify it to recognise A_1^*. The resulting NDFA N will accept its input whenever it can be broken into several pieces, with N_1 accepting each piece.

Construct N like N_1 with additional ϵ arrows returning to the start state from the accept states.
When processing gets to the end of a piece that N_1 accepts, N has the option of jumping back to the start state to try and read another piece that N_1 accepts. N must also accept ϵ as this is always a member of A_1^*.

Could add the start state to the set of accept states, but this may also add undesired strings to the recognised language, rather than just ϵ. Instead, add a new start state (which is also an accept state), and has an ϵ arrow to the old start state. This adds ϵ to the language without adding anything else.

Proof. Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognise A_1. Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognise A_1^*.

1. $Q = Q_1 \cup \{q_0\}$
   The states of N are those of N_1, plus the new start state

2. q_0 is the new start state

3. $F = F_1 \cup \{q_0\}$
   The accept states are the old accept states, plus the new start state

4. Define δ so that for any $q \in Q$ and any $A \in \Sigma_\epsilon$:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \epsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ and } a = \epsilon \\ \{q_1\} & q = q_0 \text{ and } a = \epsilon \\ \emptyset & q = q_0 \text{ and } a \neq \epsilon \end{cases}$$

4 Regular expressions

Use regular operations to build up expressions describing languages, which are called **regular expressions**.

Say that R is **regular expression** if R is:

1. a for some a in alphabet Σ,
   Regular expression a represents language $\{a\}$
2. ϵ,
   Regular expression ϵ represents language $\{\epsilon\}$
3. $\emptyset$,
   Represents the empty language
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

Note that ϵ represents the language containing a single string–the empty string–whereas $\emptyset$ represents the language that does not contain any strings.

For convenience, let R^+ be shorthand for RR^*. R^* has all strings that are 0 or more concatenations of strings from R. R^+ has all strings that are 1 or more concatenations of strings from R. $R^+ \cup \epsilon = R^*$.

Let R^k be shorthand for the concatenation of k R's with each other.

As R_1 and R_2 are always smaller than R, a circular definition is avoided (defining the notion of a regular expression in terms of itself). Defining regular expressions in terms of smaller regular expressions avoids circularity – this is an inductive definition.

To distinguish between a regular expression R and the language it describes, use $L(R)$ to describe the language of R.

The order of operator precedence is star, concatenation, then union.

4.1 Examples

Example 1. Σ describes the language consisting of all strings with length 1 over alphabet Σ.

Example 2. Σ^* describes the language consisting of all strings over Σ.

Example 3. Σ^*1 is the language containing all strings that end in a 1.

Example 4. $(0\Sigma^*) \cup (\Sigma^*1)$ consists of all strings that start with a 0 or end with a 1.

Example 5. $(\Sigma\Sigma\Sigma)^* = \{w | \text{the length of } w \text{ is a multiple of } 3\}$

Example 6. $1^*(01^+)^* = \{w|\text{every 0 in } w \text{ is followed by at least one 1}\}$

Example 7. $1^*\emptyset = \emptyset$. Concatenating the empty set to any set yields the empty set.

Example 8. $\emptyset^* = \{\epsilon\}$. The star operation puts together any number of strings from the language to get a string in the result. As the language is empty, the star operation can put together 0 strings, giving only the empty string.

Example 9. $R \cup \emptyset = R$. Adding the empty language to any language will not change it.

Example 10. $R \circ \epsilon = R$. Joining the empty string to any string will not change it.

Example 11. $R \cup \epsilon$ may not equal R. For example, if $R = 0$, $L(R) = \{0\}$, but $L(R \cup \epsilon) = \{0, \epsilon\}$.

Example 12. $R \circ \emptyset$ may not equal R. For example, if $R = 0$, $L(R) = \{0\}$, but $L(R \circ \emptyset) = \emptyset$.

Example 13. $(0 \cup \epsilon)1^* = 01^* \cup 1^*$. Expression $0 \cup \epsilon$ describes language $\{0, \epsilon\}$, so the concatenation operation adds either 0 or ϵ before every string in 1^*.

Example 13. $\Sigma^*001\Sigma^* = \{w|w \text{ contains the string 001 as a substring}\}$.

Example 14. $01 \cup 10 = \{01, 10\}$.

Example 15. $(\Sigma\Sigma)^* = \{w|w \text{ is a string of even length}\}$. The length of a string is the number of symbols it contains.

4.2 Equivalence with finite automata

Regular expressions and finite automata have equivalent descriptive power. Any regular expression can be converted into a finite automaton that recognises the language it describes, and vice versa.

A regular language is a language recognised by some finite automaton.

Theorem. A language is regular if and only if some regular expression describes it.

Proof.

($\rightarrow$) Suppose a regular expression R describing some language A. Convert R into an NDFA recognising A; if an NDFA recognises A, it is a regular language.
There are six cases in the formal definition of regular expressions:

1. $R = a$ for some $a \in \Sigma$. $L(R) = \{a\}$.
   NDFA $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, where $\delta(q_1, a) = \{q_2\}$ and $\delta(r, b) = \emptyset$ for $r \neq q_1$ or $b \neq a$, recognises $L(R)$.

2. $R = \epsilon$. $L(R) = \{\epsilon\}$.
   NDFA $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$ where $\delta(r, b) = \emptyset$ for any r and b, recognises $L(R)$.

3. $R = \emptyset$. $L(R) = \emptyset$.
   NDFA $N = (\{q\}, \Sigma, \delta, q, \emptyset)$, where $\delta(r, b) = \emptyset$ for any r and b, recognises $L(R)$.

4. $R = R_1 \cup R_2$.
   The class of regular languages is closed under union.
   Construct NDFA for R from those for R_1 and R_2 and closure under union construction.

5. $R = R_1 \circ R_2$.
   The class of regular languages is closed under concatenation.
   Construct NDFA for R from those for R_1 and R_2 and closure under concatenation construction.

6. $R = R_1^*$.
   The class of regular languages is closed under star.
   Construct NDFA for R from R_1 and closure under star construction.

($\leftarrow$) Idea: because A is regular, it is accepted by a DFA. Describe a procedure for converting DFAs intk equivalent regular expressions.

This part of the proof is given in the next section on generalised nondeterministic finite automata.

5 Generalised nondeterministic finite automata

Generalised nondeterministic finite automata (GNFAs) are nondeterministic finite automata where transition arrows may have any regular expressions as labels, instead of just members of the alphabet or ϵ.

GNFAs read blocks of symbols from the input, not necessarily just one symbol at a time. A GNFA moves along a transition arrow connecting two states by reading a block of symbols from the input, which themselves constitute a string

described by the regular expression on that arrow.

GNFAs are nondeterministic so may have several different ways to process the same input string.

A GNFA accepts its input if its processing can cause the GNFA to be in an accept state at the end of the input.

Formally, a **generalised nondeterministic finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_{start}, q_{accept})$ where:

1. Q is the finite set of states
2. Σ is the input alphabet
3. $\delta : (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \to R$ is the transition function
4. q_{start} is the start state
5. q_{accept} is the accept state

R is the collection of all regular expressions over alphabet Σ. If $\delta(q_i, q_j) = R$, the arrow from state q_i to state q_j has regular expression R as its label.

The domain of the transition function is $(Q - \{q_{accept}\}) \times (Q - \{q_{start}\})$ as an arrow connects every state to every other state, except that no arrows come from q_{accept} or go to q_{start}.

A GNFA **accepts** a string w in Σ^* if $w = w_1 w_2 ... w_k$, where each w_i is in Σ^* and a sequence of states $q_0, q_1, ..., q_k$ exists such that:

1. $q_0 = q_{start}$ is the start state
2. $q_k = q_{accept}$ is the accept state
3. for each i, $w_i \in L(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$; R_i is the expression on the arrow from q_{i-1} to q_i

5.1 Special form

For convenience, require that GNFAs have a special form meeting the following conditions:

- The start state has transition arrows going to every other state, but no arrows coming in from any other state
- There is only a single accept state, which has arrows coming in from every other state, but no arrows going to any other state.
- The accept state is not the same as the start state.
- Except for the start and accept states, one arrow goes from every state to every other state, and also from each state to itself.

5.2 Converting DFAs to GNFAs

A DFA can be converted to a GNFA in the special form:

- Add a new start state with an ϵ arrow to the old start state
- Add a new accept state with ϵ arrows from the old accept states
- If any arrows have multiple labels, or if there are multiple arrows going between the same two states in the same direction, replace each with a single arrow whose label is the union of the previous labels
- Add arrows labelled $\emptyset$ between states that had no arrows. This doesn't change the language recognised as a transition labelled with $\emptyset$ can never be used.

5.3 Converting GNFAs to regular expressions

Suppose a GNFA with k states. As a GNFA must have a start and an accept state, which must be different from each other, $k \geq 2$.

- If $k > 2$, construct an equivalent GNFA with $k - 1$ states. This step can be repeated on the new GNFA until it is reduced to two states.
- If $k = 2$, the GNFA has a single arrow going from the start state to the accept state – the label of this arrow is the equivalent regular expression.

Cruicially, constructing an equivalent GNFA with one less state (when $k > 2$) is done by selecting a state, removing it from the machine, and repairing the remainder of the machine so that the same language is still recognised.

Call the removed state q_{rip}. After removing this state, alter the regular expressions that label the remaining arrows to repair the machine. These labels should compensate the absence of the state and add back the lost computations.

The new label, going from state q_i to state q_j, is a regular expression that would take the machine from q_i to q_j, either directly or through q_{rip}.

`CONVERT(G):`

1. Let k be the number of states in G
2. If $k = 2$, G must consist of a start state, an accept state, and a single arrow connecting them and labelled with a regular expression R. Return the expression R.
3. If $k > 2$, select any state $q_{rip} \in Q$ different from q_{start} and q_{accept}. Let G' be the GNFA $(Q', \Sigma, \delta', q_{start}, q_{accept})$, where $Q' = Q - \{q_{rip}\}$ for any $q_i \in Q' - \{q_{accept}\}$.
   For any $q_j \in Q' - \{q_{start}\}$, let $\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4)$, for $R_1 = \delta(q_i, q_{rip})$, $R_2 = \delta(q_{rip}, q_{rip})$, $R_3 = \delta(q_{rip}, q_j)$, and $R_4 = \delta(q_i, q_j)$.

4. Compute `CONVERT(G')` and return this value.

Now, prove that `CONVERT` returns a correct value.

Claim. For any GNFA G, `CONVERT(G)` is equivalent to G.

Proof. Prove this by induction on k, the number of states of the GNFA.

Basis: If $k = 2$, G can only have a single arrow, which goes from the start state to the accept state. The regular expression label on this arrow describes all strings that allow G to get to the accept state – thus, this expression is equivalent to G.

Induction step: Assume the claim is true for $k-1$ states – use this assumption to prove the claim is true for k states. First show that G and G' recognise the same language.

Suppose G accepts an input w. Then, in an accepting branch of the computation, G enters a sequence of states $q_{start}, q_1, q_2, ..., q_{accept}$.

- If none of them are the removed state q_{rip}, G' also accepts w as each of the new regular expressions labelling the arrows of G' contains the old regular expression as part of a union.
- If q_{rip} appears, removing each run of consecutive q_{rip} states forms an accepting computation for G'. The states q_i and q_j bracketing a run have a new regular expression on the arrow between them – this describes all strings taking q_i to q_j via q_{rip} on G. Thus, G' accepts w.

Suppose G' accepts an input w. As each arrow between any two states q_i and q_j in G' describes the collection of strings taking q_i to q_j in G, either directly or through q_{rip}, G must also accept w. Therefore, G and G' are equivalent.

The induction hypothesis states that when the algorithm recursively calls itself on input G', the result is a rgular expression equivalent to G', as G' has $k-1$ states. Thus, this regular expression is equivalent to G, and the algorithm is correct.

6 Nonregular languages

Finite automata are limited in power – can prove that certain languages cannot be recognised by finite automata.

Consider language $B = \{0^n 1^n | n \geq 0\}$. A DFA recognising B would need to remember how many 0s (not limited) have been seen so far as it reads the input; there are an unlimited number of possibilities to track. This can't be done with finitely many states.

6.1 Pumping lemma

The pumping lemma states that all regular languages have a special property. The property states that all strings in the language can be *pumped* if they are at least as long as the *pumping length*, a certain special value.

That means that each such string contains a section that can be repeated any number of times, with the resulting string remaining in the language.

If a language does not have this property, it is guaranteed not to be regular.

Pumping lemma. If A is a regular language, there is a number p (the pumping length), where if s is any string in A of at least length p, then s may be divided into three pieces $s = xyz$, satisfying:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Note that $|s|$ represents the length of string s, y^i means that i copies of y are concatenated together, and $y^0 = \epsilon$.

Either x or z may be ϵ, but condition 2 says that $y \neq \epsilon$. Without this condition, the thorem would be trivially true.

Condition 3 states that the pieces x and y togrther have length at most p. This is an extra technical condition that is occasionally useful when proving certain languages to be nonregular.

Proof idea. Let M be a DFA recognising A, and pumping length p be the number of states in M. Show that any string s in A of length at least p can be broken into three pieces xyz, satisfying the above conditions.

- No strings of length at least p – theorem is vacuously true. The three conditions hold for all strings of length at least p if there aren't any such strings.
- If s in A has length at least p, consider the sequence of states that M goes through when computing with s. Starts with q_1, the start state, until it reaches the end of s in state q_k. With s in A, M accepts s, so q_k is an accept state.
  Let n be the length of s. Then, the sequence of states $q_1, ..., q_k$ has length $n + 1$. As $n \geq p$, $n + 1 > p$, the number of states in M.
  Thus, the sequnece must contain a repeated state, per the pigeonhole principle (if p pigeons are placed in fewer than p holes, some hole has to have more than one pigeon in it)

Proof. Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA recognising A, and p be the number of states of M. Let $s = s_1s_2...s_n$ be a string in A of length n, where $n \geq p$.

Let $r_1, ..., r_{n+1}$ be the sequence of states that M enters while processing s, so

$r_{i+1} = \delta(r_i, s_i)$ for $1 \leq i \leq n$. This sequence has length $n+1$, which is at least $p+1$. Among the first $p+1$ elements in the sequence, two must be the same state by the pigeonhole principle. Let the first of these be r_j adn the second r_l. As r_l occurs among the first $p+1$ places in a sequence starting at r_1, $l \leq p+1$.

Let $x = s_1...s_{j-1}$, $y = s_j...s_{l-1}$, and $z = s_l...s_n$. As x takes M from r_1 to r_j, y takes M from r_j to r_j, and z takes M from r_j to r_{n+1}, which is an accept state, M must accept xy^iz for $i \geq 0$. As $j \neq l$, $|y| > 0$. As $l \leq p+1$, $|xy| \leq p$.

Thus, all conditions of the pumping lemma have been satisfied.

6.2 Showing a language is not regular

- Assume that a language B is regular in order to obtain a contradiction
- Use the pumping lemma to guarantee the existence of a pumping length p, such that all strings of length p or greater in B can be pumped
- Find a string s in B that has length p or greater but cannot be pumped
- Show that s cannot be pumped by considering all ways of dividing it into x, y, and z–taking condition 3 into account if convenient–and, for each such division, finding a value i where $xy^iz \notin B$.
  This step often involves grouping the various ways of dividing s itno several cases and individually analysing them.
- The existence of s contradicts the pumping lemma if B were regular. Hence, B cannot be regular.

Finding s often needs creative thinking, and searching through several candidates before finding one that works. Typically, try members of B that seem to exhibit the "essence" of B's nonregularity.