Automata, Formal Languages, and Turing Machines

Dr. Emre Sermutlu

Contents

Preface

The subject matter of this book is about the theoretical foundations of computer science. It gives us an idea about what *any* computer program can or cannot do.

I have organized the material in 14 chapters, called weeks. I tried to make life easier for both students and instructors. The number of topics included, the balance of theory and practice, the difficulty of exercises are just right in my opinion. But I am aware of how subjective this is. After all, I decided to write this book because none of the existing textbooks were "just right" for me.

I want to thank all my students at Çankaya University with whom I shared this journey. I couldn't have completed this work without their helpful comments and criticisms.

Special thanks are to my lifelong friend Prof. Ali Aydın Selçuk and two of my students, Oğuz Küçükcanbaz and Süha Tanrıverdi for their extraordinary support.

All feedback about the book is welcome! You can reach me at `emresermutlu@gmail.com`

Dr. Emre SERMUTLU

September 26, 2020.

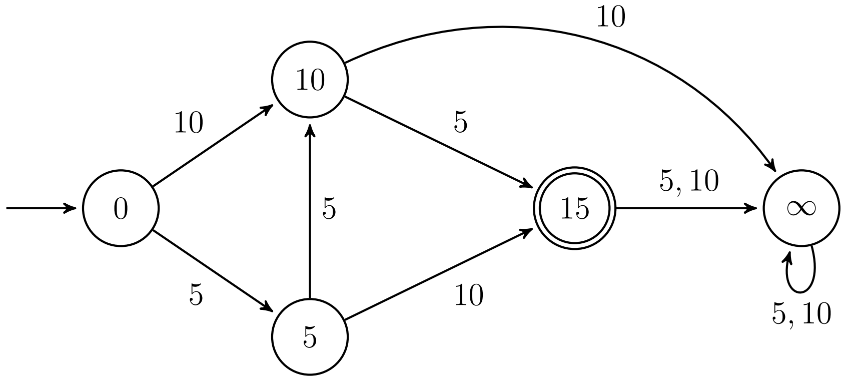
Ankara – Turkey.

Week 1

Finite Automata

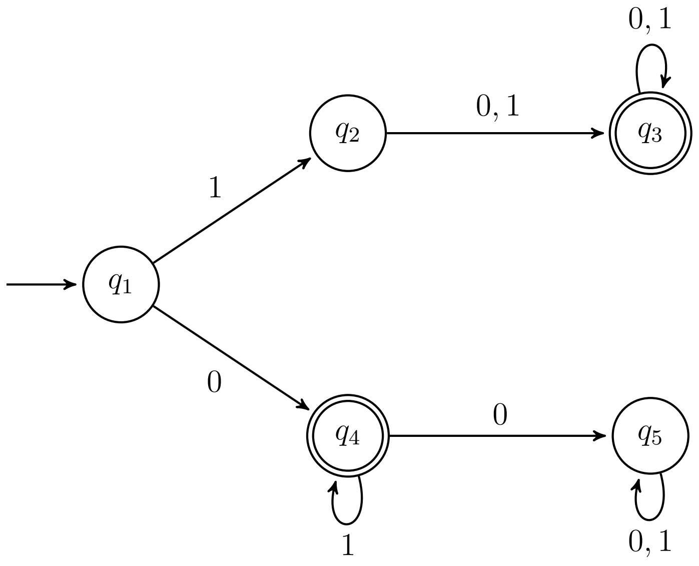
1.1 Introduction

Imagine a vending machine that accepts only $\$5$ and $\$10$ bills and sells a product priced $\$15$. After you insert bills, you press a button. Then, the machine either accepts your request and gives the product or rejects your request and returns the money.



Clearly, if we stop at the state 15, the machine accepts. Otherwise, it rejects our request.

An **alphabet** is a finite set of symbols.
A **string** over an alphabet is a finite sequence of symbols. A string that has zero length is called the empty string and denoted by ε.
A **language** is a set of strings.
The state diagram of a typical **Deterministic Finite Automaton (DFA)** looks like this:



- Here, the alphabet is: $\Sigma = \{0, 1\}$.
- The states of the automaton are: q_1, q_2, q_3, q_4 and q_5.
- The start state is q_1 and the accept states are q_3 and q_4.
- The arrows are called transitions. Note that there is one and only one arrow for each state and each symbol in the alphabet.
- The input to this automaton is a string over the alphabet, such as 010001. The output is accept or reject.

By trial and error, we can see that the automaton accepts the strings:

$$0, 01, 11, 10, 111, 10000, 011, 111111, \text{ etc.}$$

and it rejects the strings:

$$1, 00, 010, 0000, 010101, 0001, \text{ etc.}$$

We can describe the set of all strings accepted by this machine as:

- If the string starts with 1, its length must be two or greater.
- If it starts with 0, it must continue with 1's until the end. (Include no other 0.)

Note that all the states that are not accept states are reject states. Also, when we enter q_5 we cannot get out and it is a reject state, so we will call it a **trap state**.

Transition Tables: An alternative to schematic representation of a DFA is to give its transition table:

	0	1
$\rightarrow q_1$	q_4	q_2
q_2	q_3	q_3
$\circledcirc\ q_3$	q_3	q_3
$\circledcirc\ q_4$	q_5	q_4
q_5	q_5	q_5

Given the state the machine is in now and a symbol from the alphabet, we can find the next state in a unique way. In other words, this computation is **deterministic**.

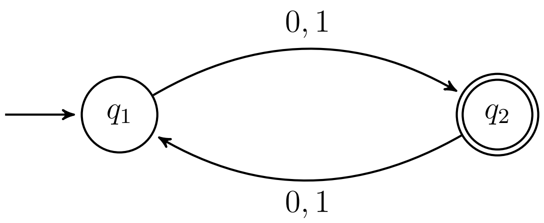
1.2 The Formal Definition of DFA

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

- Q is a finite set called the states,
- Σ is a finite set called the 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.

If A is the set of all strings that automaton M accepts, we say that A is the language of automaton M and write $L(M) = A$. This means M recognizes A.

Example 1–1: What language does the following automaton recognize?

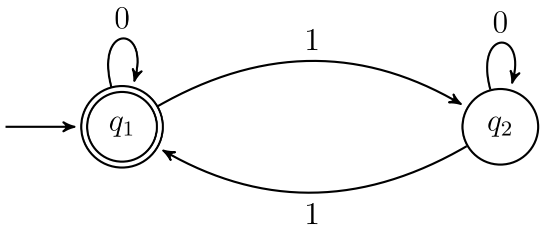


Solution: We may try some simple strings and note that it accepts 0 and 1, rejects 00 and 10, accepts 101, etc.

For any symbol from the input, the DFA changes state. That means, it oscillates between even and odd cases.

It accepts all strings with an odd number of symbols.

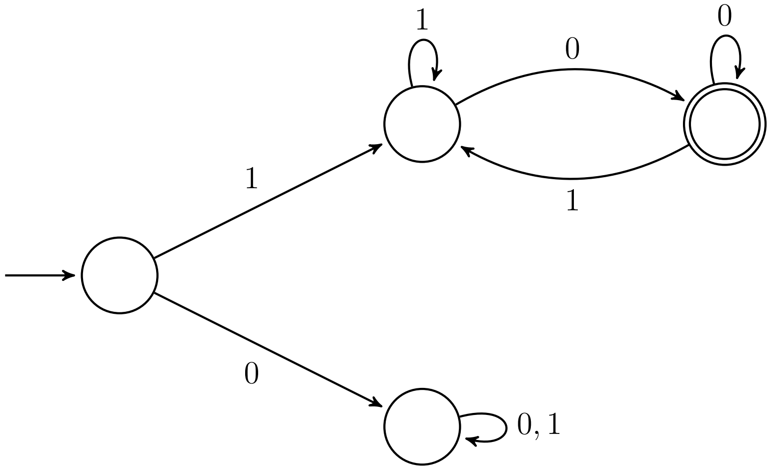
Example 1–2: What language does the following automaton recognize?



Solution: 0's do not make any difference, they do not change the state of the DFA. So this is all about 1's.

This DFA accepts all strings containing an even number of 1's.

Example 1–3: What language does the following automaton recognize?



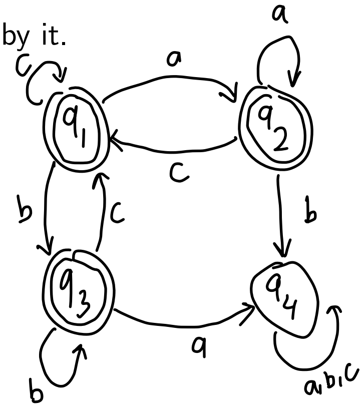
Solution: Note that if the string starts with 0, it is rejected. If it ends with 1, it is also rejected. So this DFA recognizes the language made of strings starting with 1 and ending with 0.

Example 1–4: A DFA with 4 states over the input alphabet $\Sigma = \{a, b, c\}$ is given by the following transition table.

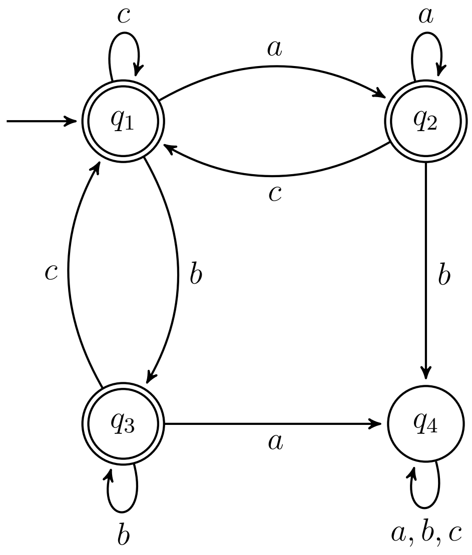
a) Find the state diagram of the DFA.

b) Verbally describe the language recognized by it.

	a	b	c
$\rightarrow \circledcirc\, q_1$	q_2	q_3	q_1
$\circledcirc\, q_2$	q_2	q_4	q_1
$\circledcirc\, q_3$	q_4	q_3	q_1
q_4	q_4	q_4	q_4



Solution:



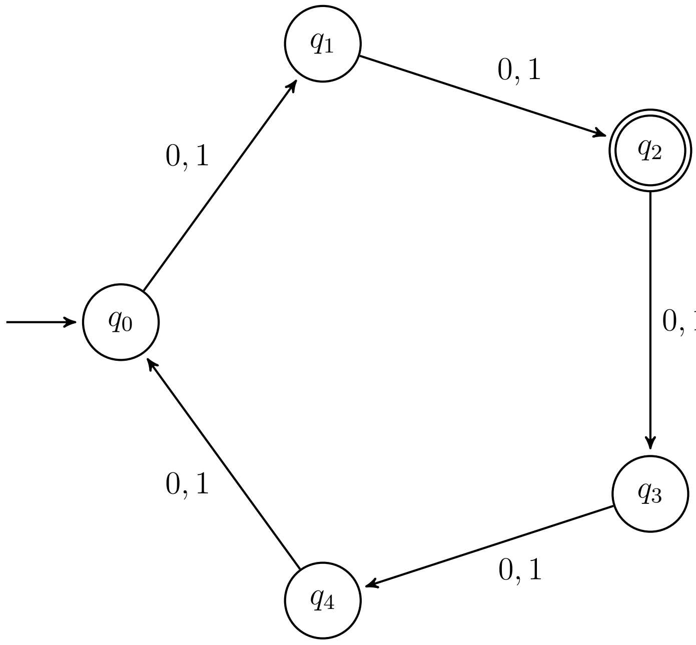
Note that q_4 is a trap state. There is no ab or ba in the strings of this language.
In other words, there must always be at least one c between a and b.

Example 1–5: Find a finite automaton that recognizes the language

$$L = \{w \mid \text{the number of digits of } w \text{ is } 2 \text{ modulo } 5\}$$

over the alphabet $\Sigma = \{0, 1\}$.

Solution: We need to count the symbols modulo 5. This means we need 5 states and only one of them will be the accept state.



There is no difference between 0 and 1, we just consider the number of symbols. But we can also design a DFA that counts 0's or 1's in a similar way.

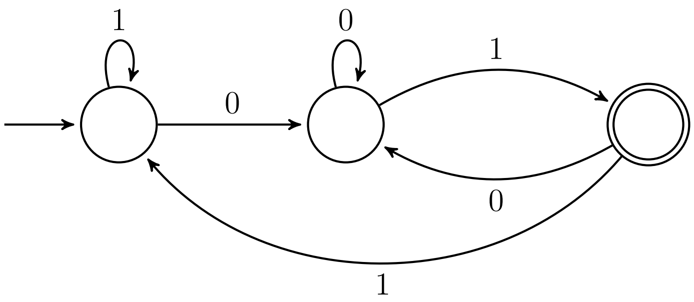
Example 1–6: Find a finite automaton that recognizes the language

$$L = \{w \mid w \text{ ends with } 01\}$$

over the alphabet $\Sigma = \{0, 1\}$.

Solution: The automaton should check for the string 01.

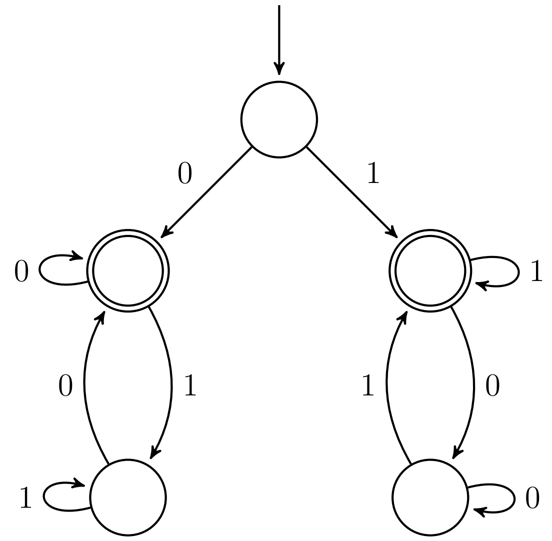
- If we receive 0 in the start state, we move to the intermediate state.
- If we receive 1 when we are in the intermediate state, we move to the accept state.
- If there is no other symbol in the string, we accept.
- If we receive 0 when in the accept state, we have to move to the intermediate state.
- If we receive 1 when in the accept state, we have to move to the start state.



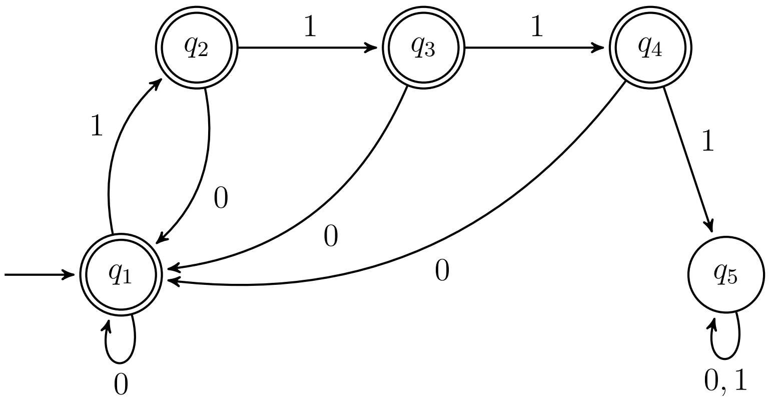
EXERCISES

Describe the languages over alphabet $\Sigma = \{0, 1\}$ recognized by the following DFAs:

1–1)

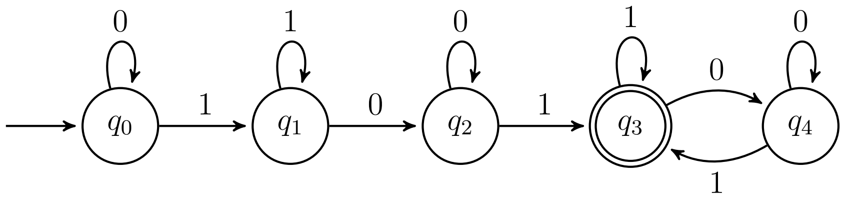


1–2)

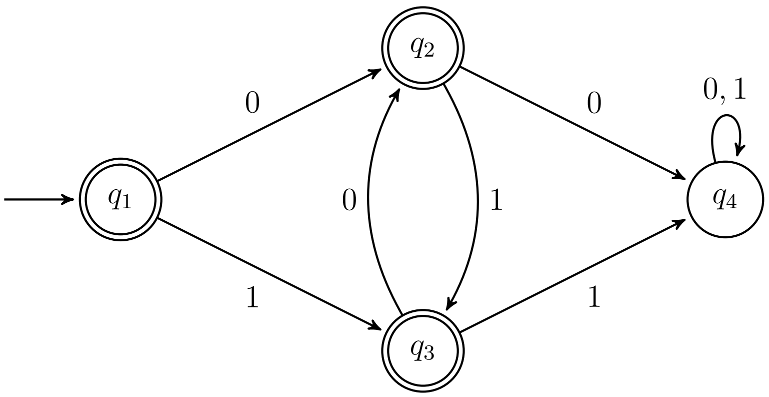


Describe the languages over alphabet $\Sigma = \{0, 1\}$ recognized by the following DFAs:

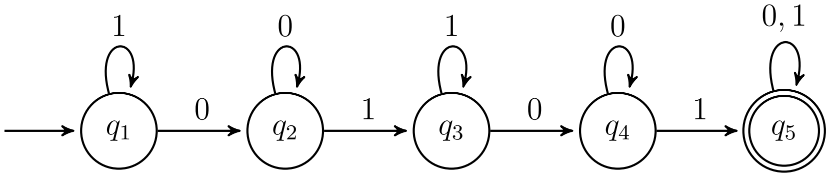
1–3)



1–4)

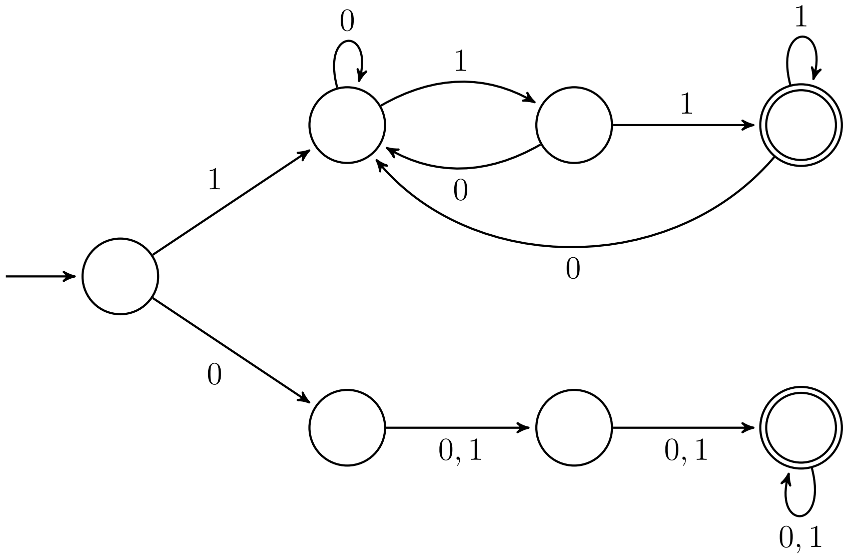


1–5)

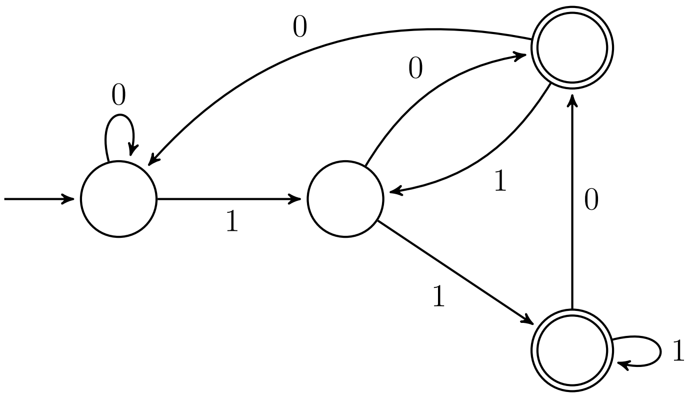


Describe the languages over alphabet $\Sigma = \{0, 1\}$ recognized by the following DFAs:

1–6)

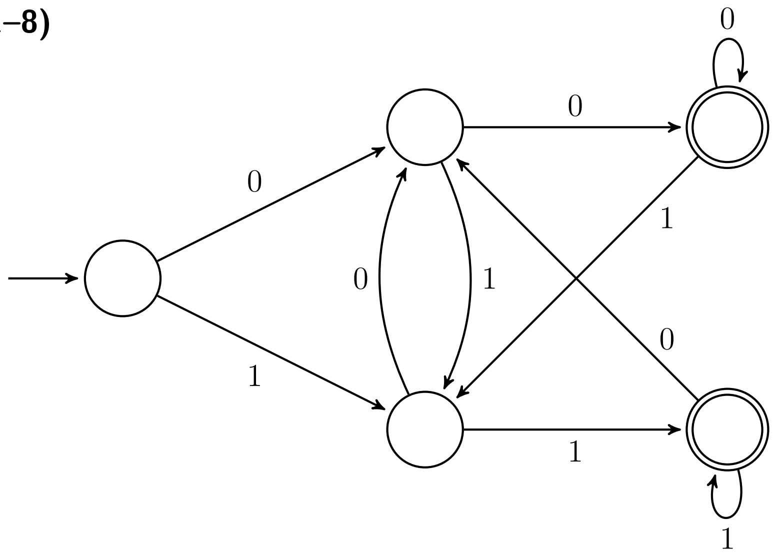


1–7)

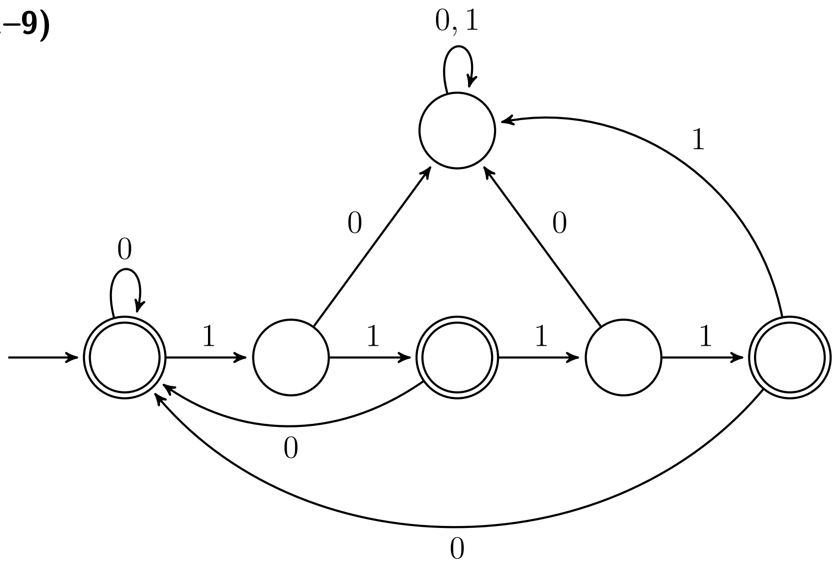


Describe the languages over alphabet $\Sigma = \{0, 1\}$ recognized by the following DFAs:

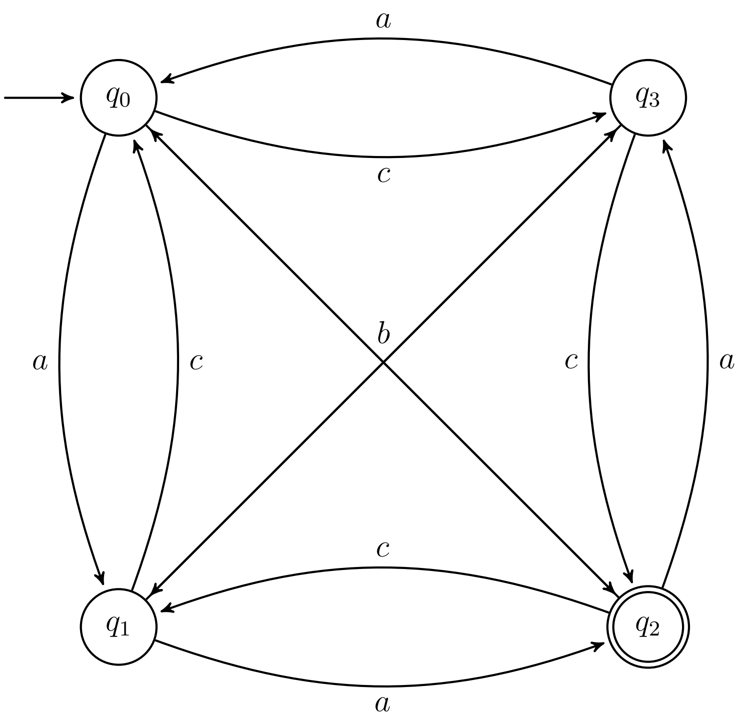
1–8)



1–9)



1–10) Describe the language over alphabet $\Sigma = \{a, b, c\}$ recognized by the following DFA:



Give the state diagram of a DFA that recognizes the language L over alphabet Σ:

1–11) $\Sigma = \{a, b\},\ L = \{w \mid w$ contains exactly two a's.$\}$

1–12) $\Sigma = \{a, b\},\ L = \{w \mid w$ contains at least two a's.$\}$

1–13) $\Sigma = \{a, b\},\ L = \{w \mid w$ contains at most two a's.$\}$

1–14) $\Sigma = \{a, b\},\ L = \{w \mid w = a^n\, b^m,$ where $n, m \geqslant 1.\}$

1–15) $\Sigma = \{a, b\},\ L = \{w \mid w = a^n\, b^m,$ where $n, m \geqslant 0.\}$

1–16) $\Sigma = \{a, b\},\ L = \{w \mid w = a^n\, b^m,$ where $n + m = 3.\}$

1–17) $\Sigma = \{0, 1\},\ L = \{w \mid w$ contains 0000 and ends with $11.\}$

1–18) $\Sigma = \{0, 1\}, \quad L = \{$Strings of length at least three whose second and third symbols are the same.$\}$

Give the state diagram of a DFA that recognizes the language L over alphabet Σ:

1–19) $\Sigma = \{a, b, c\},\ L = \Big\{w \ \Big|\ w \text{ does not contain } aa, bb \text{ or } cc.\Big\}$
For example, $\varepsilon, a, b, c, ab, cac, abc, baba, abcbcabacb, \ldots$

1–20) $\Sigma = \{0, 1\},\ L$ consists of strings that:

- Do not contain 11. (All 1's are isolated.)
- Contain 0's in sequences of two or three.

For example, $\varepsilon, 1, 00, 000, 100, 1000, 001, 0001, 00100, 0010001,$
$10001001, 00010010001, \ldots$

1–21) $\Sigma = \{0, 1\},\ L = \Big\{w \ \Big|\ w \text{ contains } 111 \text{ or } 000.\Big\}$

1–22) $\Sigma = \{a, b\},\ L = \Big\{w \ \Big|\ w \text{ contains } aaa \text{ but does not contain } bbb\Big\}.$

1–23) $\Sigma = \{0, 1\},\ L = \Big\{w \ \Big|\ w \text{ contains one or two } 0\text{'s in the first three symbols.}\Big\}$

1–24) $\Sigma = \{0, 1\},\ L$ consists of strings that:

- Are of length at least two.
- The first two symbols are the same as the last two symbols.

For example, $11, 111, 1111, 00100, 101010, 01111101, 10010010, \ldots$

ANSWERS

1–1) Binary strings that start and end with the same symbol.

1–2) Binary strings that do NOT contain 1111.

1–3) Many answers are possible. Some of them, starting with the simplest are:

- Contains 10 and ends with 1, or,
- Contains 10 and 01 and ends with 1, or,
- Contains at least two symbol changes and ends with 1, or,
- Contains at least 3 symbols, $1, 0$ and 1 in this order (but not necessarily consecutively) and ends with 1.

1–4) Binary strings that contain neither 00 nor 11.

1–5) We can describe the language in different ways:

- Contains at least two 01's, or,
- If it starts with 0, changes symbols at least three times, if it starts with 1, changes symbols at least four times.

1–6)

- String length must be 3 or greater, and,
- If a string starts with 1, it must end with 11.

1–7)

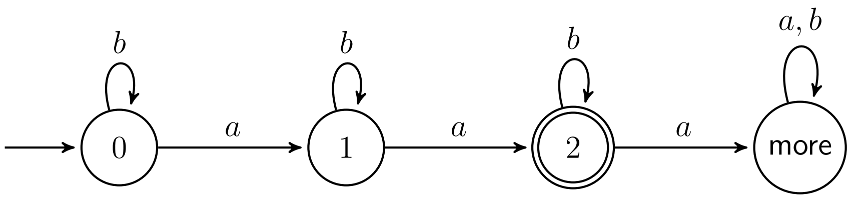
- Strings whose second symbol from the end is 1, or,
- Strings that end with 11 or 10.

1–8) Binary strings that end with 00 or 11.

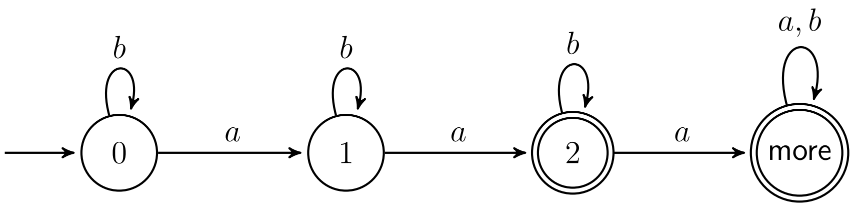
1–9) Binary strings where all 1's are in groups of two or four.

1–10) Let number of a, b, c's in the string be k, m, n. Then, the strings where $k + 2m + 3n \equiv 2 \pmod 4$ are accepted.

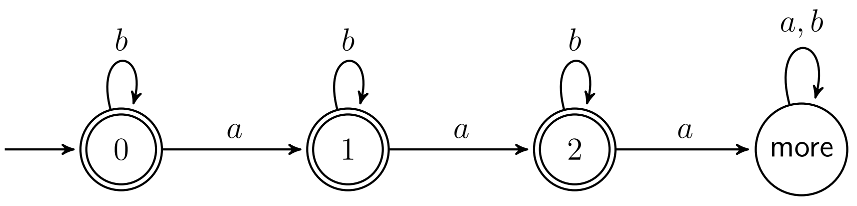
1–11)



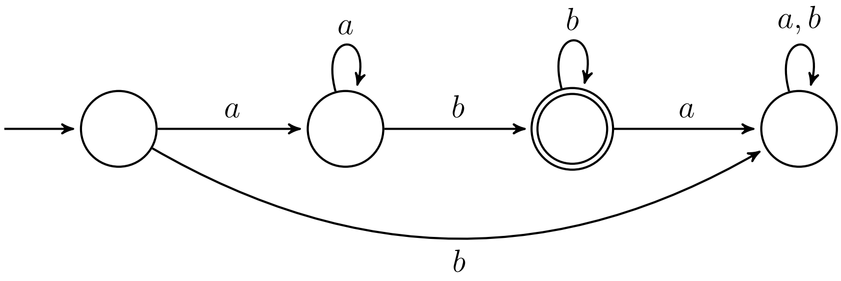
1–12)



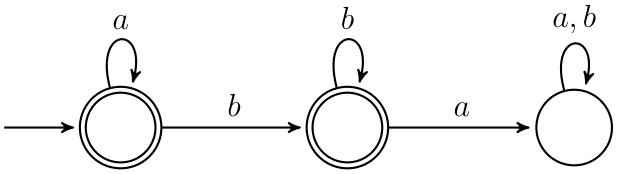
1–13)



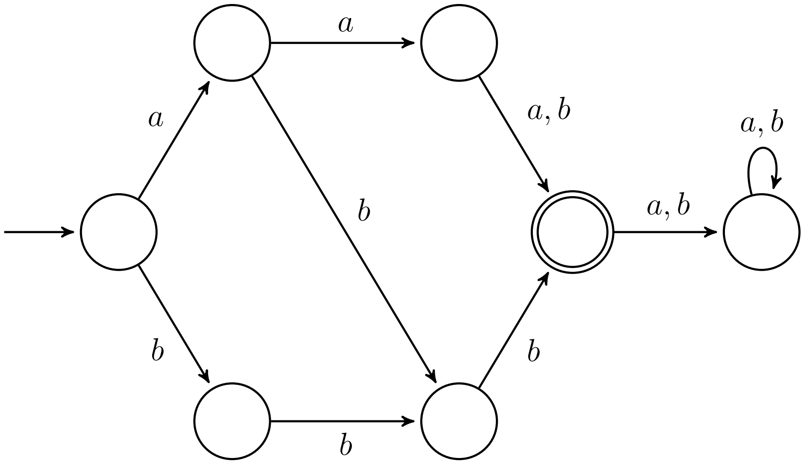
1–14)



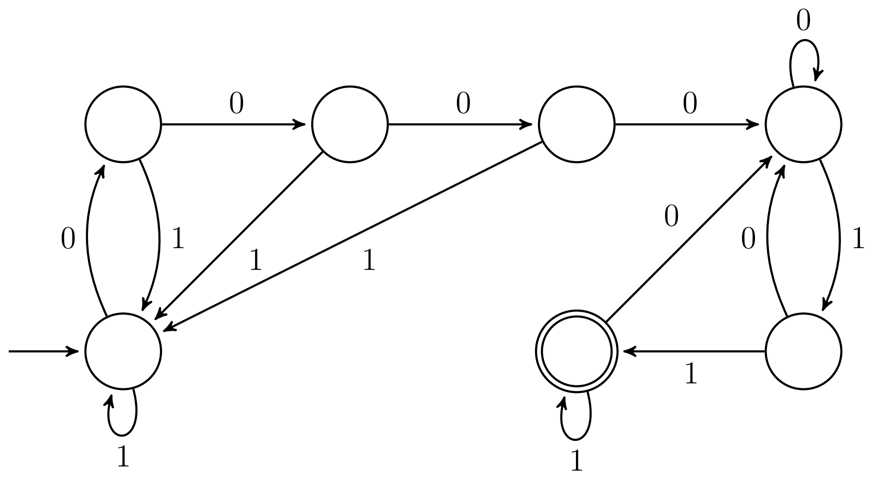
1–15)



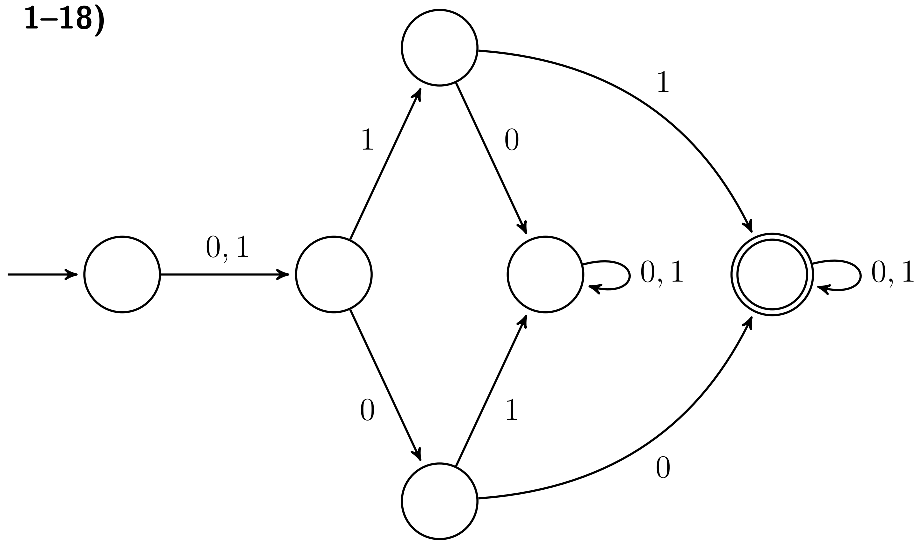
1–16)



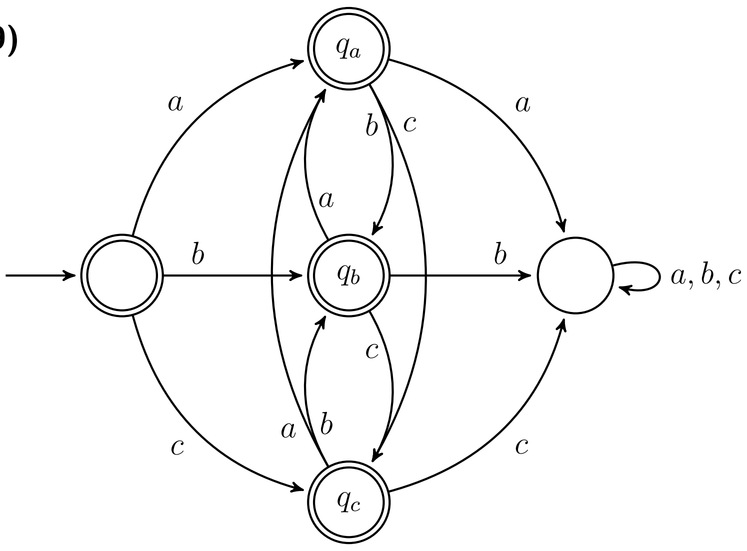
1–17)



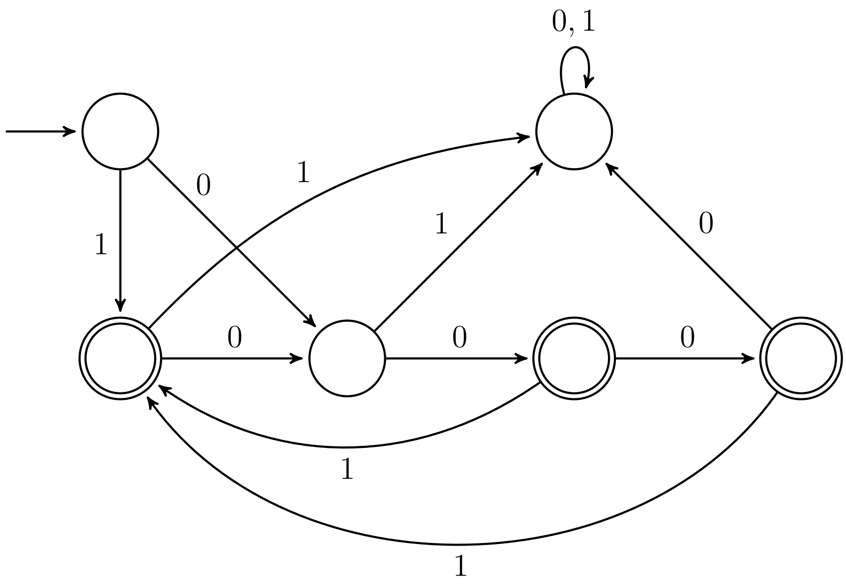
1–18)



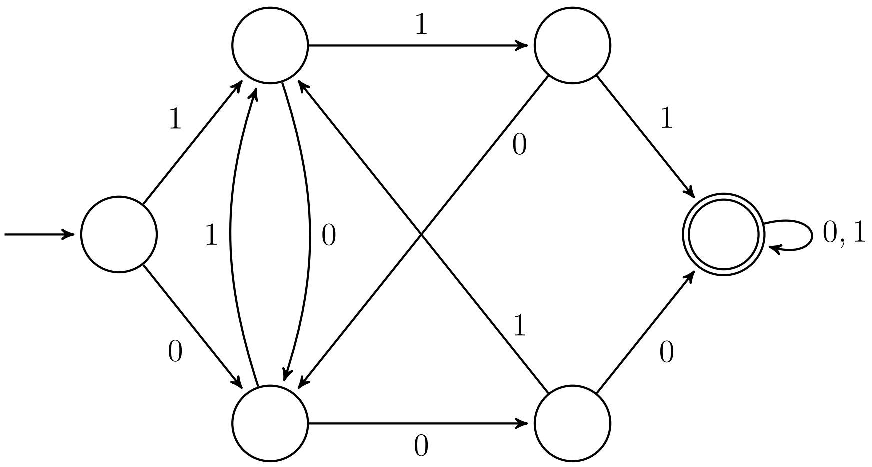
1–19)



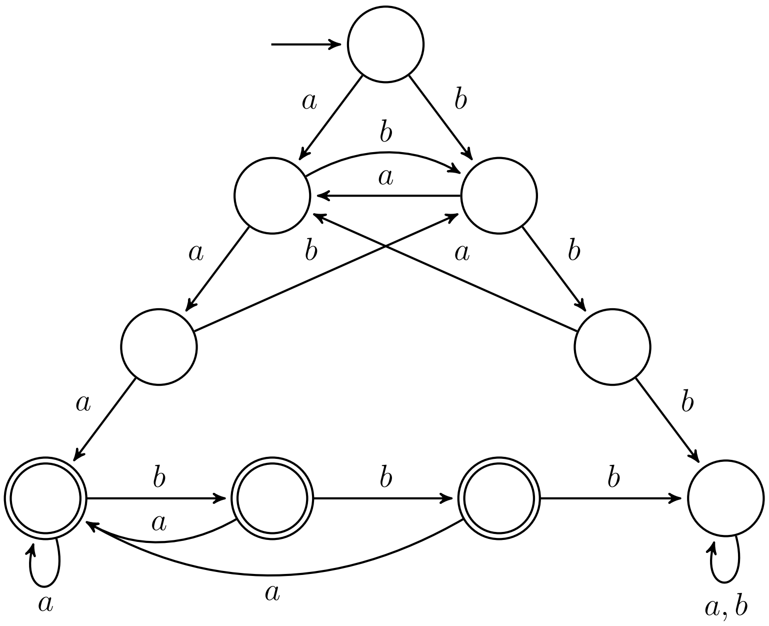
1–20)



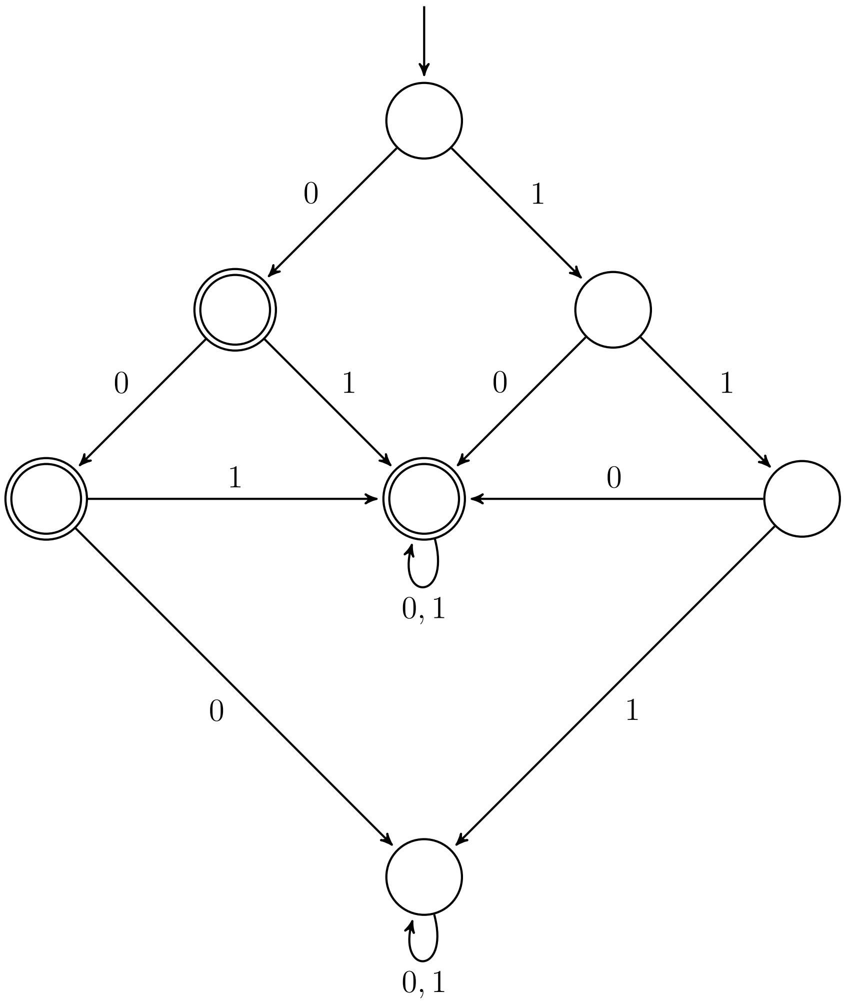
1–21)



1–22)

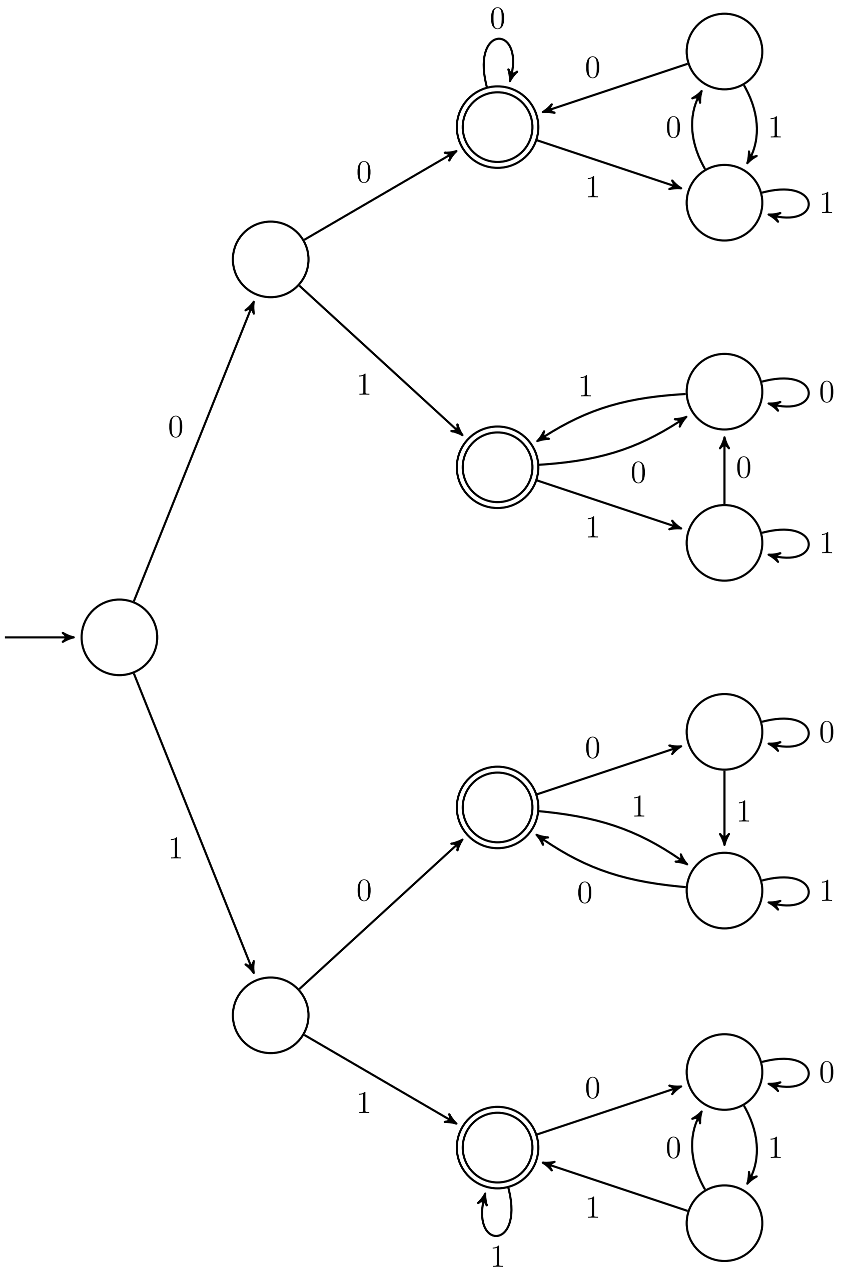


1–23)



Note that, once you get 10 or 01 in the beginning, you don't care about the third input.

1–24)



Week 2

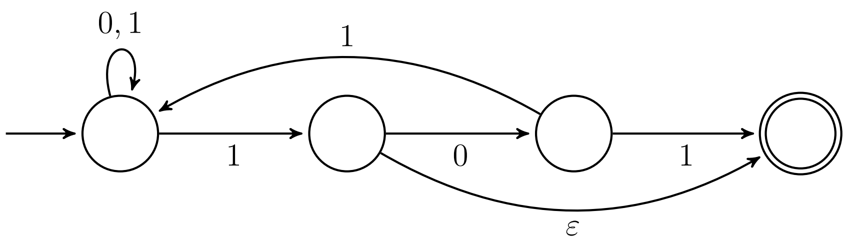
Nondeterminism

2.1 The Idea of NFA

We will use the abbreviation **NFA** for a **Nondeterministic Finite Automaton**.
An NFA is similar to a DFA except for the following differences:

- For a given symbol, a state may have more than one transition.
- For a given symbol, a state may have no transition.
- Arrows may be labeled $\varepsilon (= \text{epsilon})$.

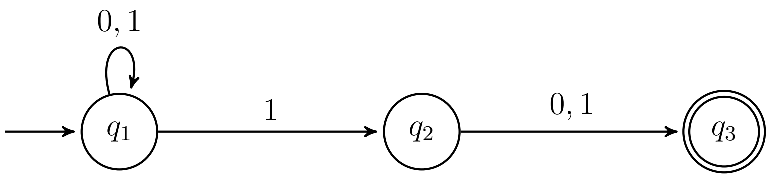
For example, this is an NFA:



Nondeterminism is a kind of parallel computation.

- If there are many arrows leaving a state for a symbol, the machine **branches** at that point. We have to follow all the resulting machines for the rest of the input. If one of the machines accepts, the NFA accepts.
- If there are no arrows leaving a state for a symbol, that machine **dies**. Or you can assume it moves into a trap state labeled $\varnothing$ and stays there.

For example, consider the NFA:



- If we receive input 1 when in state q_1, the machine splits into two machines. They have the same structure but one is in state q_1 and the other is in state q_2.

  The same thing may happen again, so we may have a large number of machines working in parallel.
- If a machine receives input 0 or 1 (in other words, any input) when in state q_3, that machine dies.

  We do not have to follow that machine any further.

You may see that the NFA above accepts all strings whose second element from the end is 1.

Compared to a DFA that does the same job, it is much simpler.

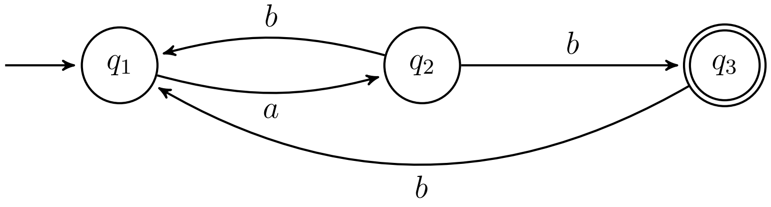
2.2 The Formal Definition of NFA

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

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta : Q \times \Sigma_\varepsilon \to 2^Q$ is the transition function,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is the set of accept states.

Σ_ε denotes $\Sigma \cup \{\varepsilon\}$ and 2^Q denotes the set of all subsets of Q.

This is the same definition as DFA except for the δ function. It takes ε as input in addition to symbols of Σ and gives a set of states as output. An empty set is also a possibility.
For example, for the following NFA,



the transition function is:

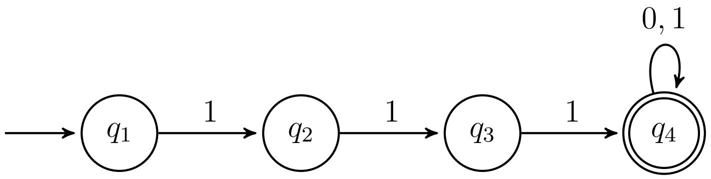
$$\begin{array}{rclcrcl} \delta(q_1, a) & = & \{q_2\} & \qquad & \delta(q_1, b) & = & \varnothing \\ \delta(q_2, a) & = & \varnothing & & \delta(q_2, b) & = & \{q_1, q_3\} \\ \delta(q_3, a) & = & \varnothing & & \delta(q_3, b) & = & \{q_1\} \end{array}$$

We omitted ε transitions because they are all empty set. Can you see which strings this NFA accepts?

Example 2–1: Give the state diagram of an NFA that recognizes the language L over alphabet $\Sigma = \{0, 1\}$ where

$$L = \Big\{ w \;\Big|\; w \text{ starts with } 111. \Big\}$$

Solution: If 0 exists among the first three symbols, that machine dies. We can express this idea very easily using an NFA.

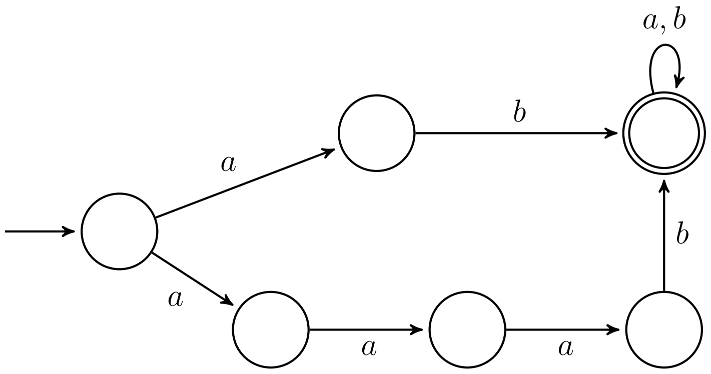


Example 2–2: Give the state diagram of an NFA that recognizes the language $A \cup B$ over alphabet $\Sigma = \{a, b\}$ where

$$A = \Big\{ w \;\Big|\; w \text{ starts with } ab. \Big\}$$

$$B = \Big\{ w \;\Big|\; w \text{ starts with } aaab. \Big\}$$

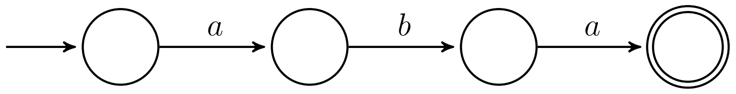
Solution: The operation of union is also easy for NFAs:



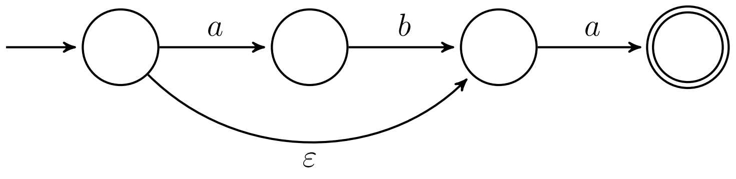
2.3 Epsilon Transitions

Using $\varepsilon-$transitions, we move between states without reading anything from the input.

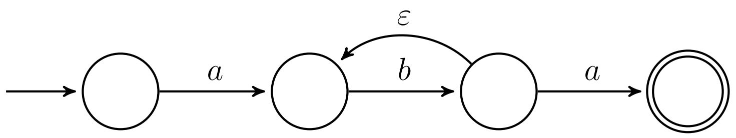
- For example, the following machine accepts the string aba and nothing else:



- But this machine accepts a in addition to aba:

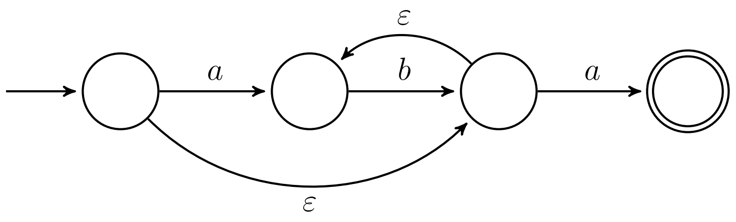


- The following machine accepts $ab^n a,\ n \geqslant 1$:



- With an additional $\varepsilon-$transition the language becomes:

$$\left\{ab^n a,\ n \geqslant 1\right\} \cup \left\{b^n a,\ n \geqslant 0\right\}$$



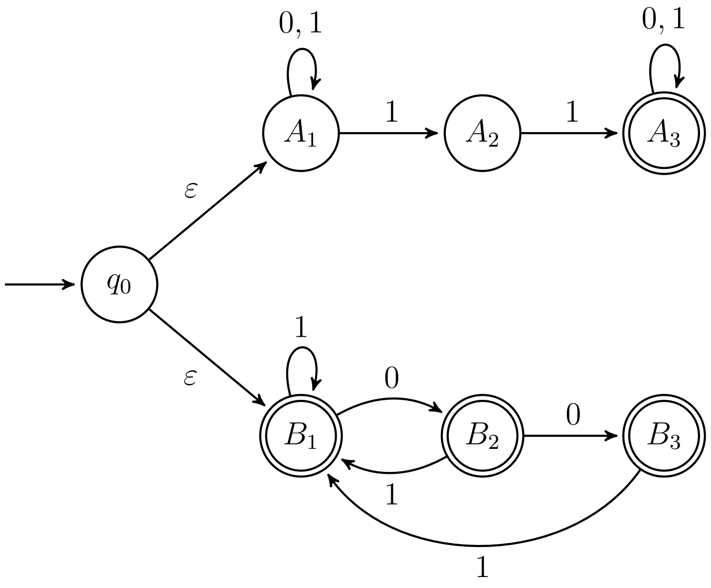
Example 2–3: Let A and B be languages over $\Sigma = \{0, 1\}$:

$$\begin{aligned} A &= \{w \mid w \text{ contains } 11.\} \\ B &= \{w \mid w \text{ does not contain } 000.\} \end{aligned}$$

Give the state diagram of an NFA that recognizes the language $A \cup B$.

Solution: The two machines that separately recognize languages A and B are very easy to construct.

Now, all we have to do is connect them in parallel using $\varepsilon-$ transitions:



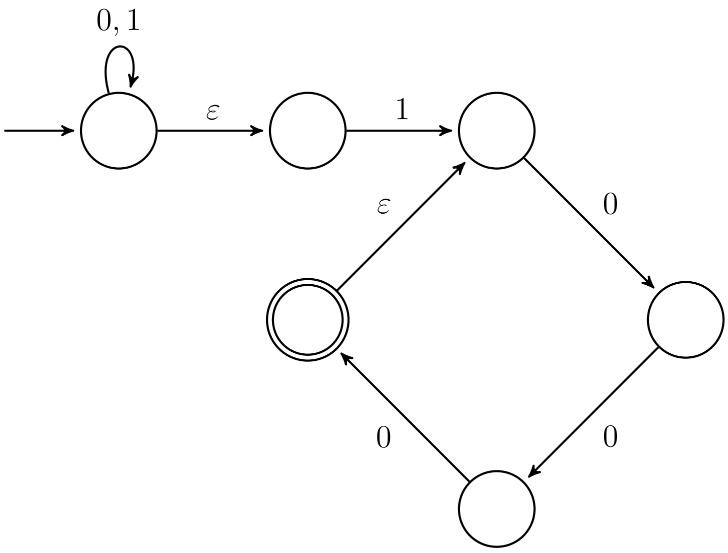
Example 2–4: Find an NFA that recognizes the following language over $\Sigma = \{0, 1\}$:

There should be at least one 1 in the string and the number of 0's after the last 1 must be $3n$ where $n \geqslant 1$.

Solution: If we know a given 1 in the string is the last 1, the rest is easy. But we do not have this kind of information.

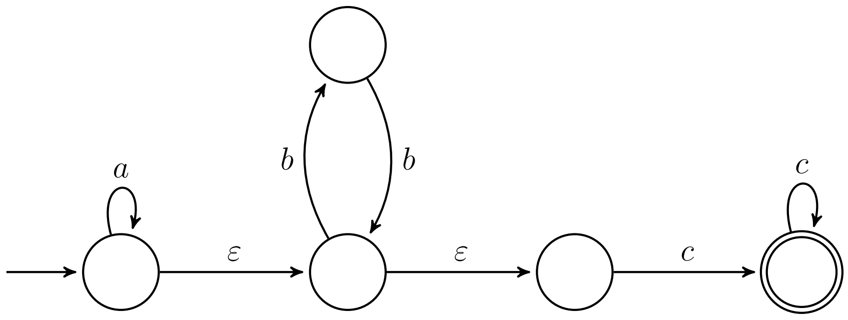
Therefore, after each 1, we assume it is the last and check 0's to see if there are a multiple of 3. But at the same time, we continue with the computation.

$\varepsilon-$transitions are a great way to do this:



We can also design a DFA recognizing the same language but it will be more complicated.

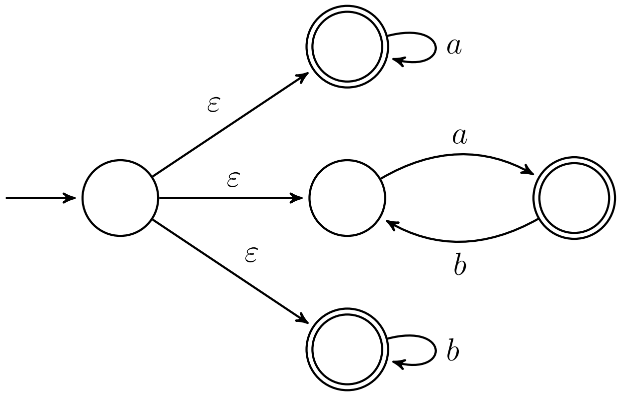
Example 2–5: Verbally describe the language recognized by the following NFA:



Solution: First, we have an arbitrary number of a's. (Possibly zero)
Then we have an even number of b's. (Possibly zero)
Then we have one or more c's.

$\Rightarrow \quad w = a^k\, b^{2m}\, c^{n+1}$ where $k, m, n \geqslant 0$

Example 2–6: Verbally describe the language recognized by the following NFA:



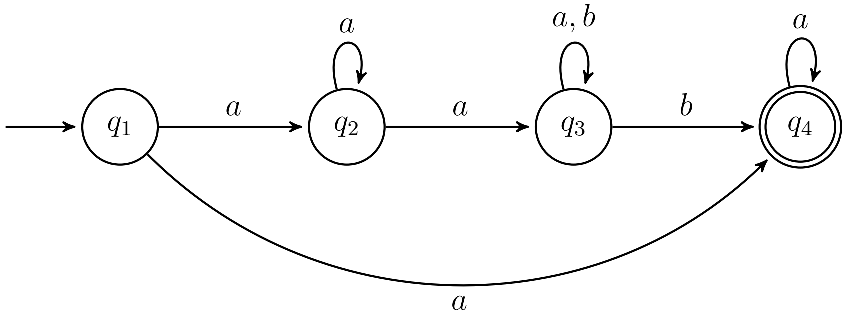
Solution: a^n or b^n or $a(ba)^n$ where $n \geqslant 0$.

2.4 NFA - DFA equivalence

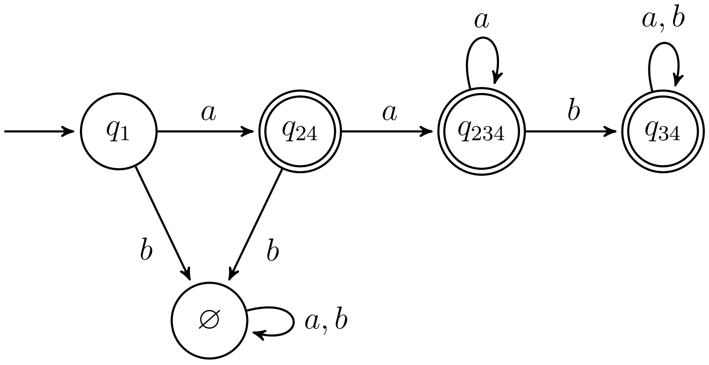
Are there languages that can be recognized by an NFA but no DFA? In other words, are NFAs more powerful than DFAs?

We may think that the answer is "*Yes!*" because NFAs have more complicated transition rules. Surprisingly, this is not this case. For any NFA, we can always find a DFA that does the same job. (But the DFA probably contains more states.)

Example 2–7: Find a DFA recognizing the same language as the following NFA:



Solution: Consider the set of all subsets of states. $2^4 = 16$ but not all of them actually occur during a computation.



Two machines are called **equivalent** if they recognize the same language.

Theorem: Every NFA has an equivalent DFA.

Sketch of Proof:

- Given an NFA with n states, construct a DFA with 2^n states. These correspond to all possible subsets of states.

  For example, if the NFA contains $\{q_1, q_2\}$ then the DFA contains $\{\varnothing, q_1, q_2, q_{12}\}$

- Consider all states that can be reached from a state through $\varepsilon-$transitions and call it the $\varepsilon-$closure of that state. For example, if $\varepsilon-$arrows exist for

  $$q_1 \rightarrow q_4,\ q_1 \rightarrow q_7 \quad \text{or} \quad q_1 \rightarrow q_4,\ q_4 \rightarrow q_7,$$

  the $\varepsilon-$closure of q_1 is q_{147}.

  Use $\varepsilon-$closures to eliminate $\varepsilon-$transitions. For example, the start state of the new machine must be the $\varepsilon-$closure of the start state of the old machine.

- If several arrows leave a state for a given input in NFA, take that state to the state representing the combined set in DFA.

  For example, if there are transitions

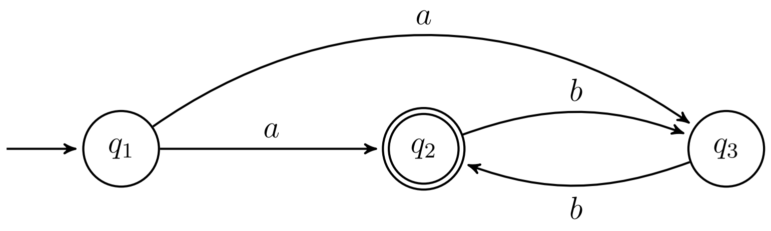
  $$q_1 \rightarrow q_2,\ q_1 \rightarrow q_4,\ q_1 \rightarrow q_5$$

  for input a in NFA, the corresponding DFA will have the transition

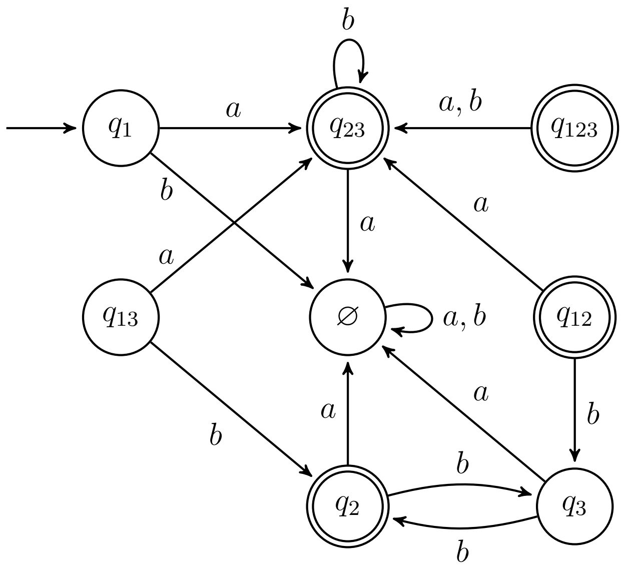
  $$q_1 \rightarrow q_{245}$$

- If no arrow leaves a state for a given input in NFA, take that state to $\varnothing$ in DFA.

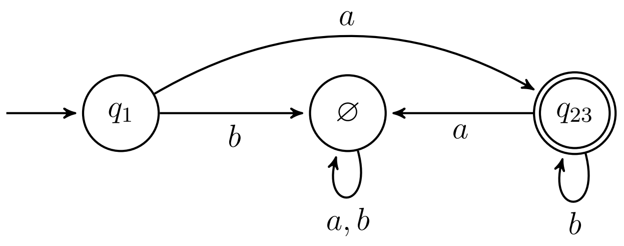
Example 2–8: Find a DFA equivalent to the following NFA over the alphabet $\Sigma = \{a, b\}$:



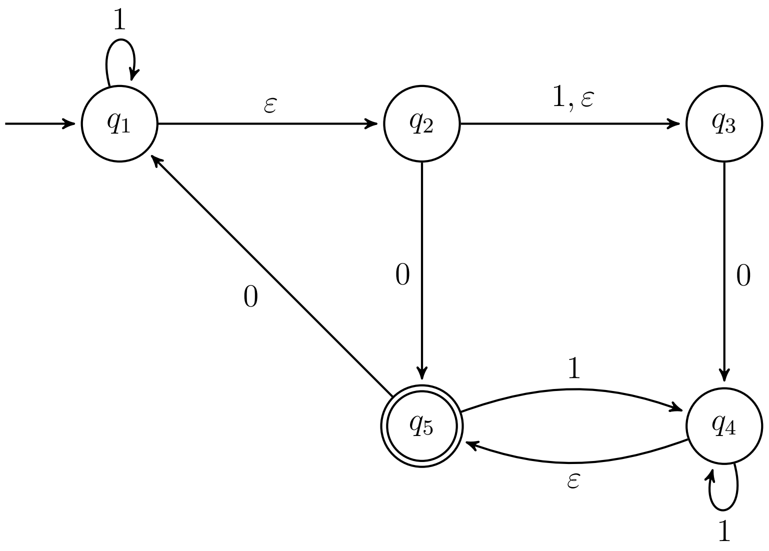
Solution: Consider a DFA with $2^3 = 8$ states and fill the transitions using sets of states:



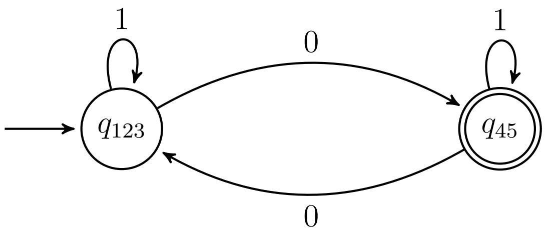
Eliminating states unreachable from start state:



Example 2–9: Find a DFA equivalent to the following NFA over the alphabet $\Sigma = \{0, 1\}$:



Solution: Using $\varepsilon-$closures of states, we obtain the simple DFA:



These machines accept strings containing an odd number of 0's.

EXERCISES

Find an NFA for each of the following languages defined over the alphabet $\Sigma = \{0, 1\}$:

2–1) Strings that start with 101.

2–2) Strings that end with 101.

2–3) Strings that start with 101 or end with 101.

Find an NFA for each of the following languages defined over the alphabet $\Sigma = \{a, b, c\}$:

2–4) Strings whose last three symbols are the same.

2–5) Strings do contain c or do not contain b.
(For example, $\varepsilon, abaaaba, aba, aa, bb, ac, accca, ccc, cacacc, \ldots$ etc.)

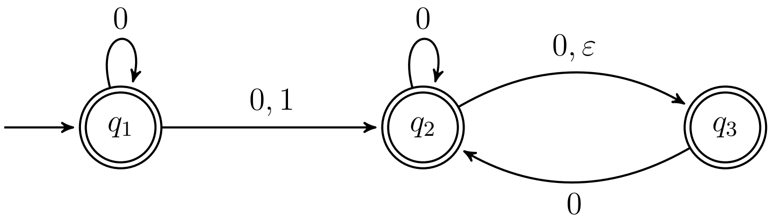
2–6) Strings that:

- Start and end with a, contain exactly two a's.
- b's and c's are alternating

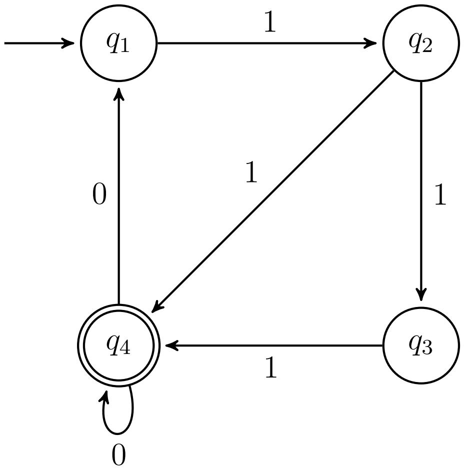
(For example, $aa, aba, aca, abca, acbca, abcbcbca, \ldots$ etc.)

Find a DFA that is equivalent to (i.e. recognizes the same language as) the given NFA:

2–7)

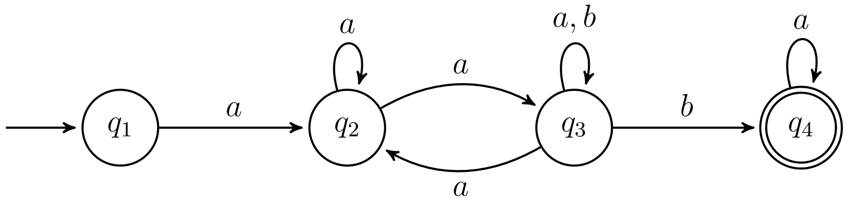


2–8)

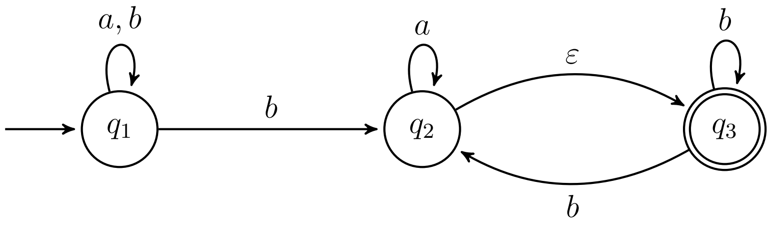


Find a DFA that is equivalent to the given NFA:

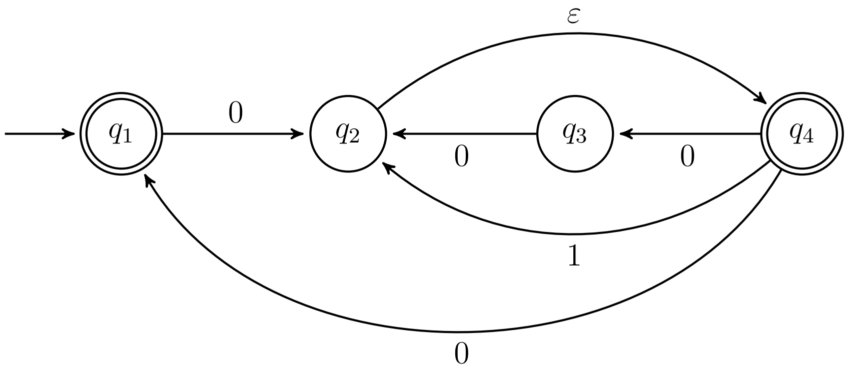
2–9)



2–10)



2–11)



Find an NFA that recognizes the language L over the alphabet Σ:

2–12) $\Sigma = \{0, 1\}$, L consists of strings that:

- Start with 01,
- End with 10,
- Are of length at least 5.

2–13) $\Sigma = \{0, 1\}$, L consists of strings that:

- Start with 0 and have odd length, or,
- Start with 1 and have length at most 3.

2–14) $\Sigma = \{a, b\}$, L consists of strings that:

- Start and end with a,
- String length must be at least two,
- Contain a's in sequences of two or more,
- Do not contain bb. (All b's are isolated.)

Find an NFA that recognizes the language L over the alphabet Σ:

(These are the same questions from Week 1. Please compare the NFAs and DFAs to understand the difference better.)

2–15) $\Sigma = \{a, b, c\},\ L = \Big\{w \;\Big|\; w$ does not contain aa, bb or $cc.\Big\}$
(For example, $\varepsilon, a, b, c, ab, cac, abc, baba, abcbcabacb, \ldots$)

2–16) $\Sigma = \{0, 1\},\ L$ consists of strings that:

- Do not contain 11. (All 1's are isolated.)
- Contain 0's in sequences of two or three.

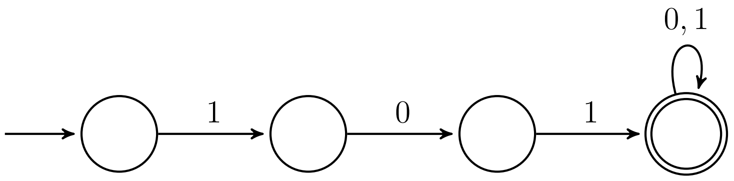
For example, $\varepsilon, 1, 00, 000, 100, 1000, 001, 0001, 00100, 0010001,$ $10001001, 00010010001, \ldots$

2–17) $\Sigma = \{0, 1\},\ L = \Big\{w \;\Big|\; w$ contains 111 or $000.\Big\}$

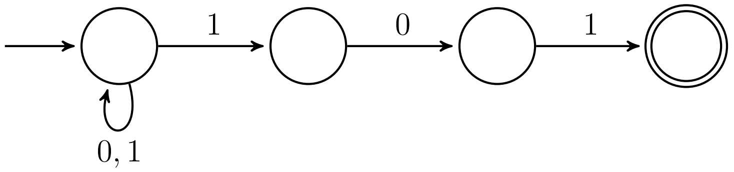
2–18) $\Sigma = \{0, 1\},\ L = \Big\{w \;\Big|\; w$ contains one or two 0's in the first three symbols.$\Big\}$

ANSWERS

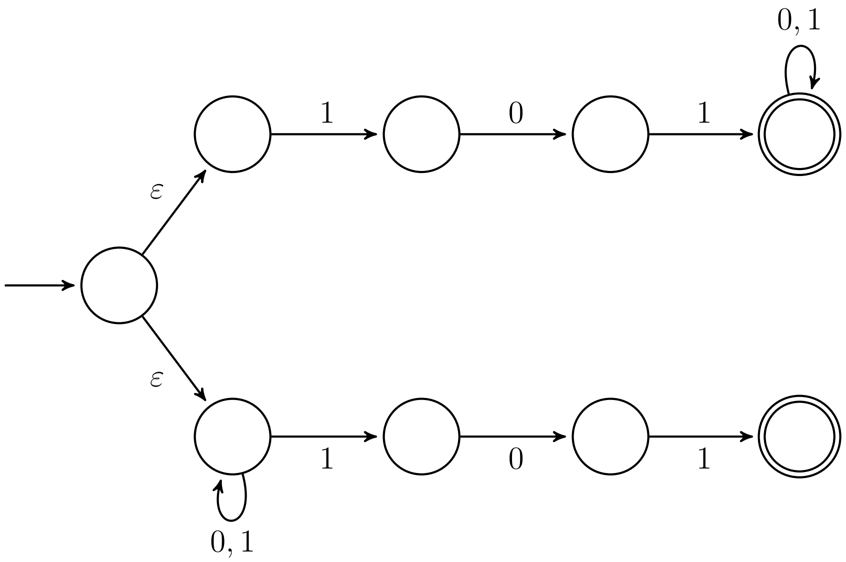
2–1)



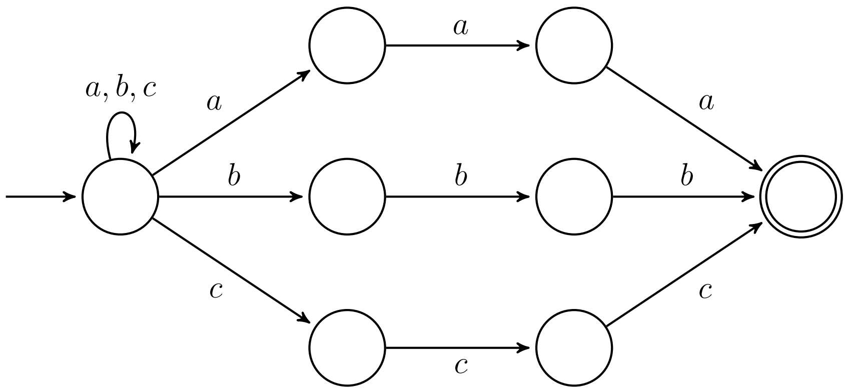
2–2)



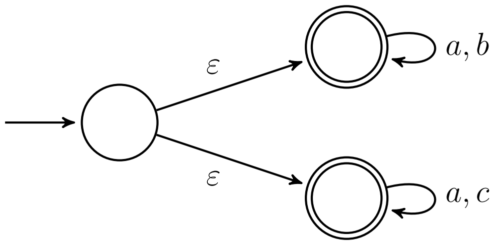
2–3)



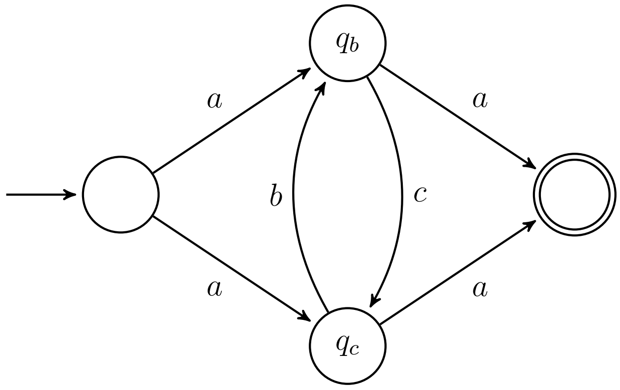
2–4)



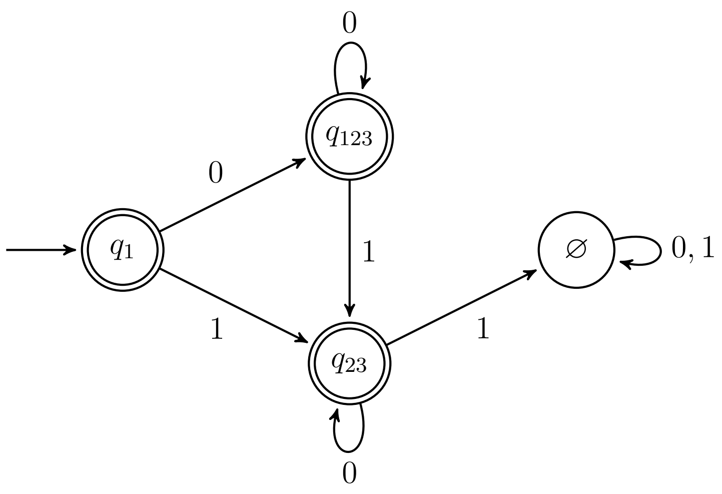
2–5)



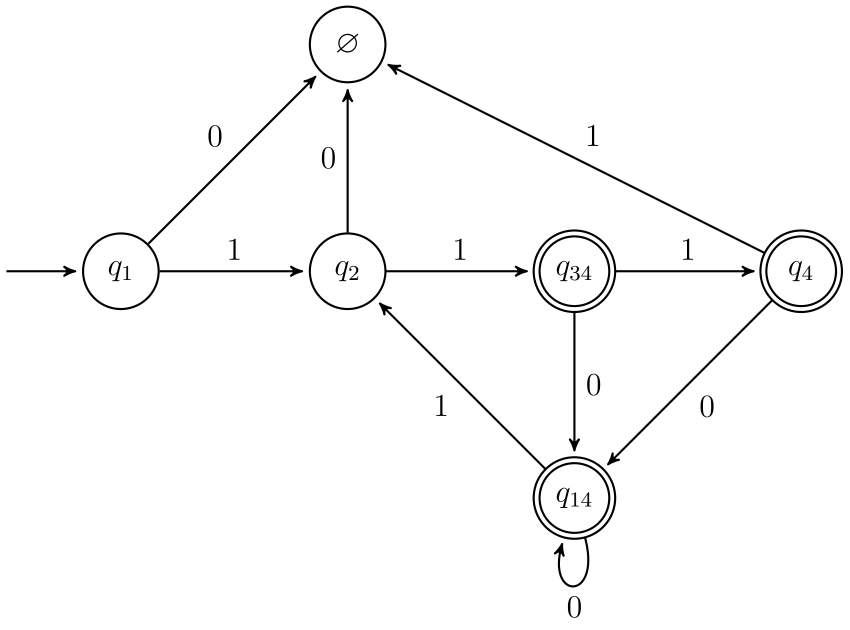
2–6)



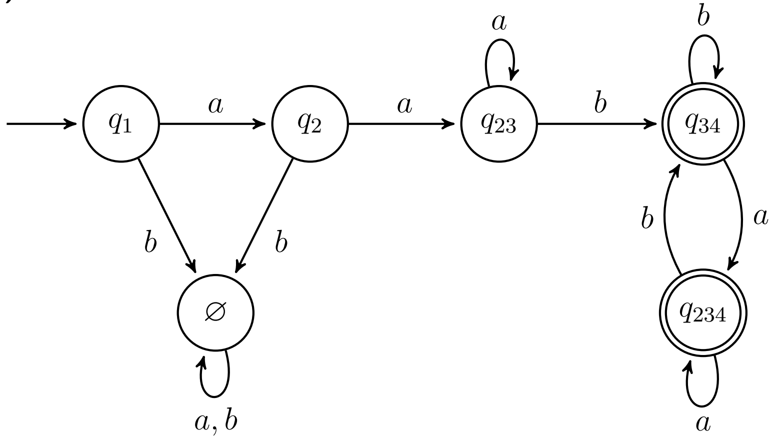
2–7)


q_1
q_{123}
q_{23}
$\varnothing$
0
0
1
1
1
0
0, 1

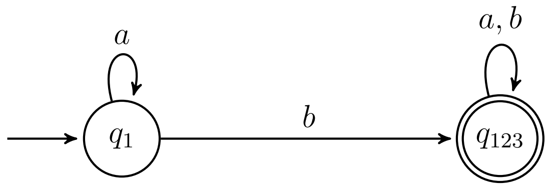
2–8)


$\varnothing$
q_1
q_2
q_{34}
q_4
q_{14}
0
0
1
1
1
1
1
0
0
0

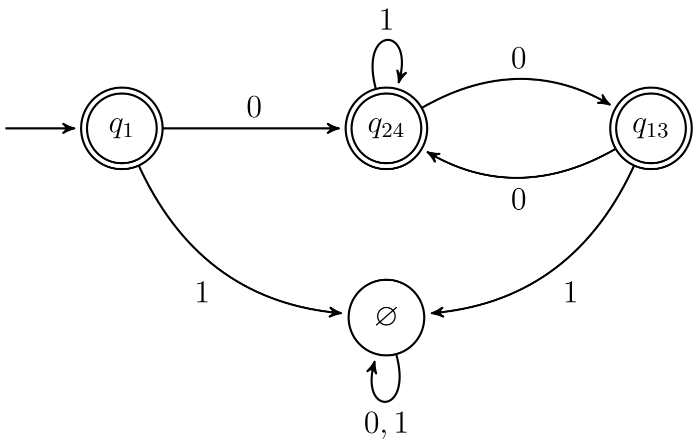
2–9)


q_1
q_2
q_{23}
q_{34}
q_{234}
$\varnothing$
a
b
a, b

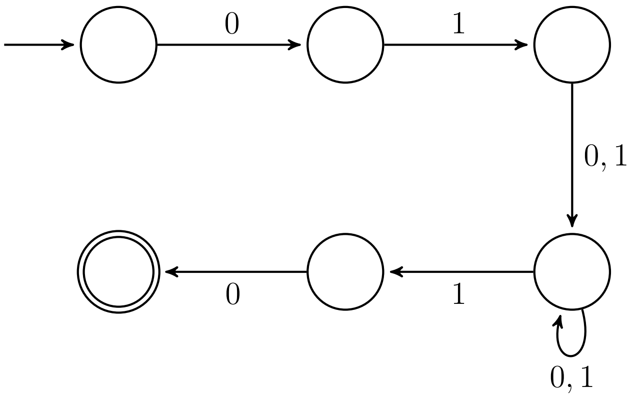
2–10)


q_1
q_{123}
a
b
a, b

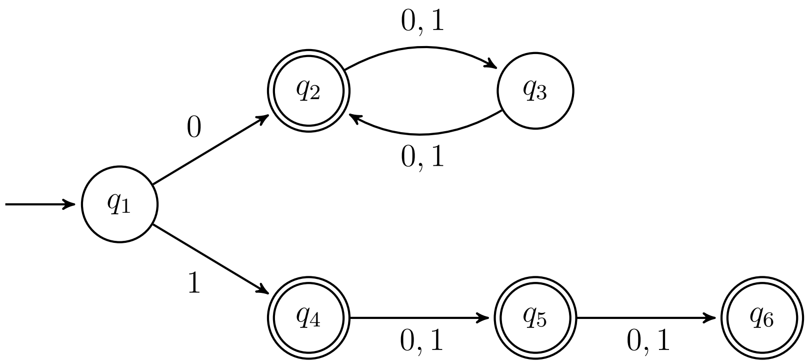
2–11)


q_1
q_{24}
q_{13}
$\varnothing$
0
1
$0, 1$

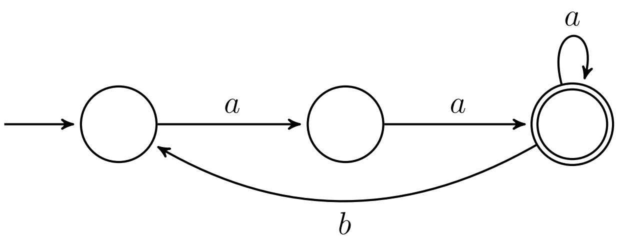
2–12)



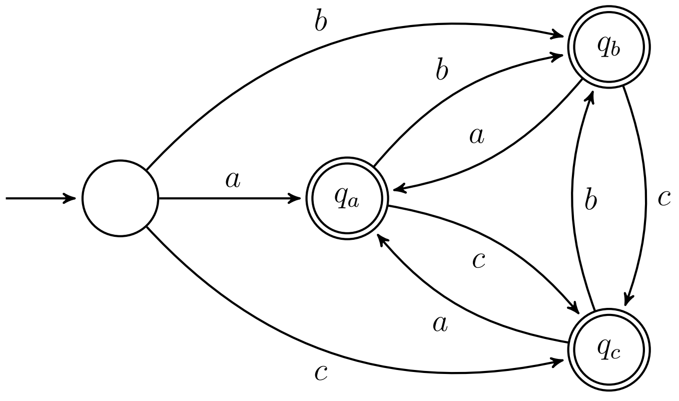
2–13)



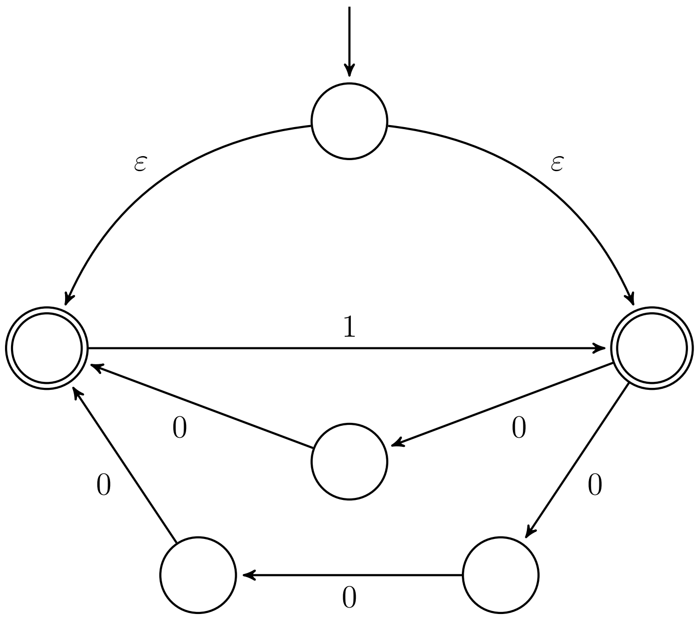
2–14)



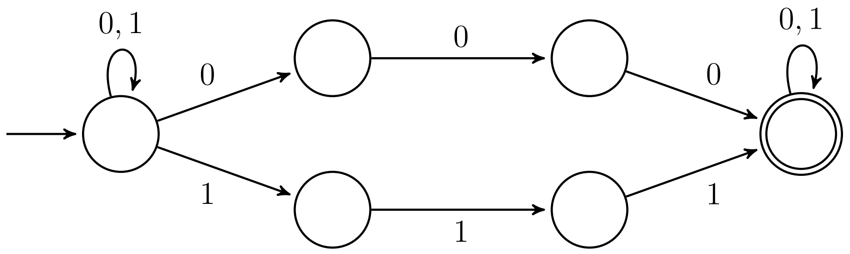
2–15) Compare with page 21, answer **1–19)**.



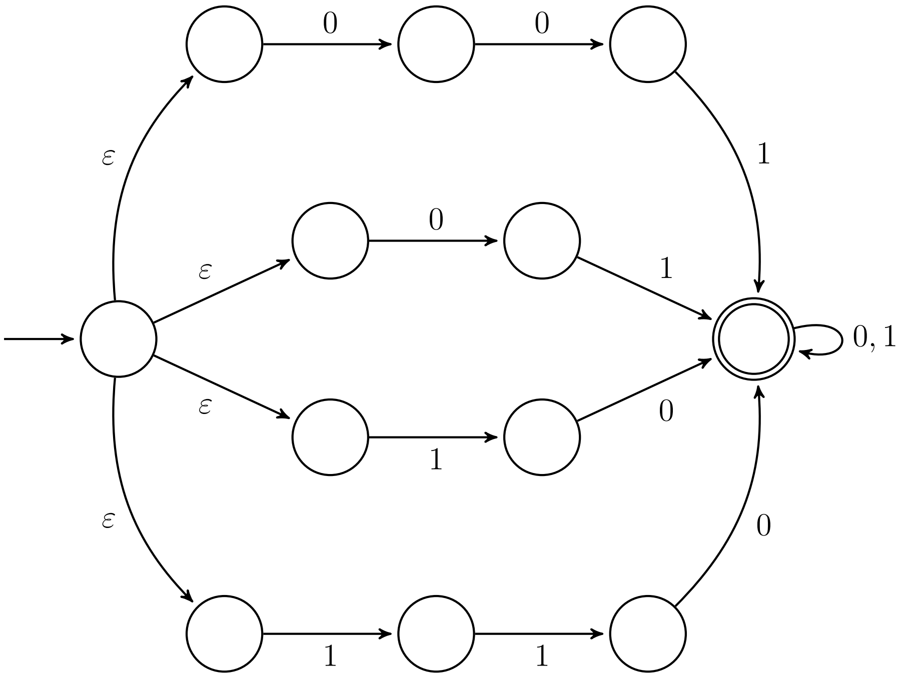
2–16) Compare with page 21, answer **1–20)**.



2–17) Compare with page 22, answer **1–21)**.



2–18) Compare with page 23, answer **1–23)**.



Week 3

Regular Languages

3.1 Operations on Regular Languages

A language is called a **regular language** if some finite automaton recognizes it. We previously proved that DFAs and NFAs are equivalent in that respect. In other words, the language is regular if we can find a DFA or an NFA that recognizes it.

For example, over the alphabet $\Sigma = \{0, 1\}$:

- $A = \{w \mid w \text{ contains an even number of } 1\text{'s}\}$
  is a regular language.
- $B = \{w \mid w \text{ contains a prime number of } 1\text{'s}\}$
  is NOT a regular language. (Can you see why?)
- $C = \{w \mid w \text{ contains an equal number of } 1\text{'s and } 0\text{'s}\}$
  is NOT a regular language. (Can you see why?)

We can obtain new languages using the following operators on given languages:

Union: $A \cup B = \{w \mid w \in A \quad \text{or} \quad w \in B\}$.

For example, if $A = \{a, ab\}$ and $B = \{bb, abba\}$ then:

$$A \cup B = \{a, ab, bb, abba\}$$

Concatenation: $A \circ B = \{vw \mid v \in A \quad \text{and} \quad w \in B\}$.

For example, if $A = \{a, ab\}$ and $B = \{bb, abba\}$ then:

$$A \circ B = \{abb, aabba, abbb, ababba\}$$

Star: $A^* = \{w_1 w_2 \dots w_k \mid k \geqslant 0 \quad \text{and each} \quad w_i \in A\}$.

Note that $k = 0$ is a possibility for star operation, therefore $\varepsilon \in A^*$ whatever A is. For example,

$$A = \{a\} \quad \Rightarrow \quad A^* = \{\varepsilon, a, aa, aaa, \dots\}$$

$$B = \{ab\} \quad \Rightarrow \quad B^* = \{\varepsilon, ab, abab, ababab, \dots\}$$

$$C = \{a, bb\} \quad \Rightarrow \quad C^* = \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, \dots\}$$

The star operator has the highest precedence. Concatenation is the second and the union is the third.

In practice, we will omit the concatenation operator $\circ$. For example, let $A = \{1\}, B = \{0\}, C = \{11\}$. Then the expression

$$AB^*A \cup C^*$$

denotes the language containing strings that:

1. Either start with 1 and end with 1 and contain an arbitrary number of 0's but no 1's between them, or,

2. Contain an even number of 1's and no 0's.

A very important problem we will consider next is:

If we start with regular languages and do these operations, do we obtain regular languages?

3.2 Closure Properties of Regular Languages

Theorem: The class of regular languages is closed under the union operation.

In other words, if A and B are regular languages, so is $A \cup B$.

Sketch of Proof: The question is that, given a DFA recognizing the language A and another one recognizing B, how can we build a third DFA that recognizes $A \cup B$?

In the beginning of the computation, we do not know if the string belongs to A or B. So our new machine must simulate the two old machines *simultaneously*. (We do not have the luxury of trying the given input in the first and then the second machine.)

Supposing given DFAs have m and n states, we can accomplish this with a new DFA having mn states. We can think of the new machine as a Cartesian product of the old ones.

1. Each state will be a combination of two states from two machines.

2. The new start state will be the combination of old start states.

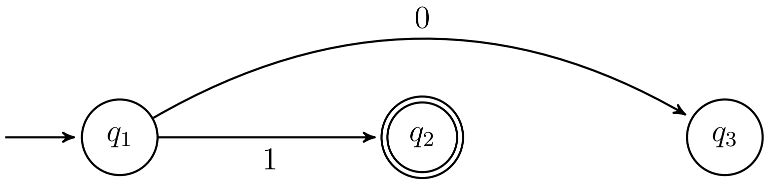
3. New accept states will be a combination of an accept state with any other state.

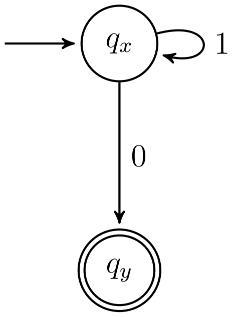
4. Transition rules will preserve the structure of both machines.

If we modify the third rule and require that the new accept states must come from a combination of accept states from both sides, this will be a machine recognizing $A \cap B$.

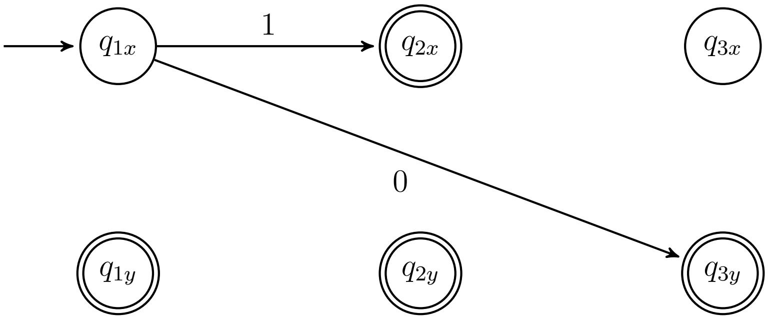
(This modified version would provide a proof that if A and B are regular languages, so is $A \cap B$.)

For example, suppose the following machines recognize the languages A and B: (These are DFAs, but for simplicity, we only show the arrows leaving the start states.)





Then the new machine that combines both is another DFA given by:



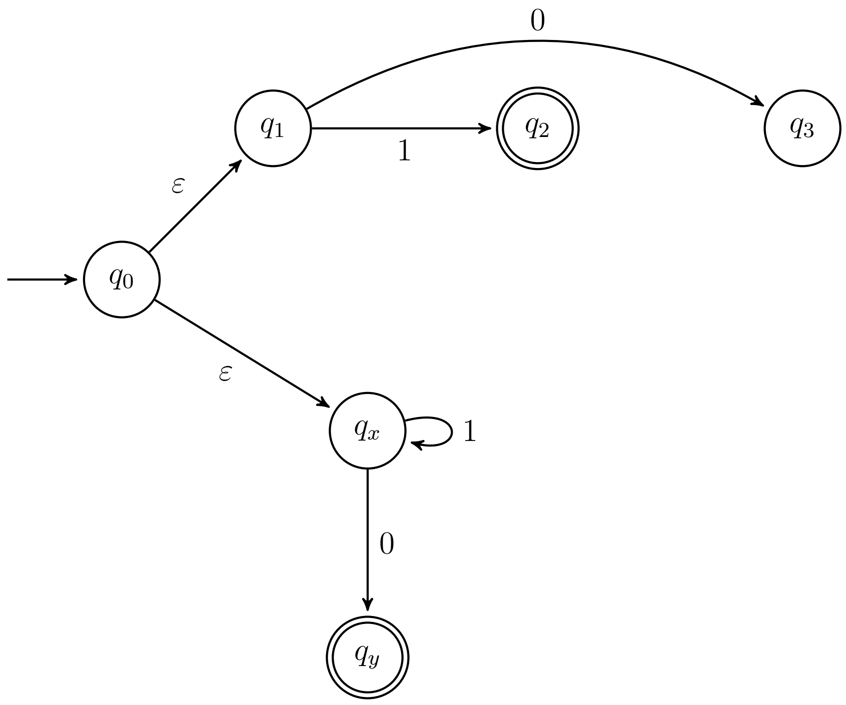
The vertical projection gives the first and the horizontal projection gives the second machine.

Alternative Proof: If we use NFAs rather than DFAs, we can obtain a new machine that recognizes the union of languages very easily.

Once again, assume we are given the previous machines that recognize the languages A and B.

Now we can form a new start state and connect to the old start states by $\varepsilon-$transitions. This will give us two machines computing in parallel. Accept states remain as accept states.

So the new NFA that recognizes $A \cup B$ is:

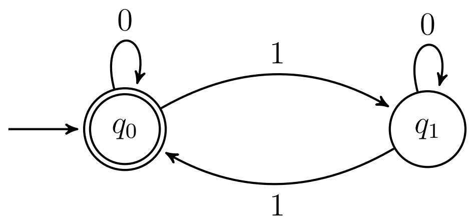


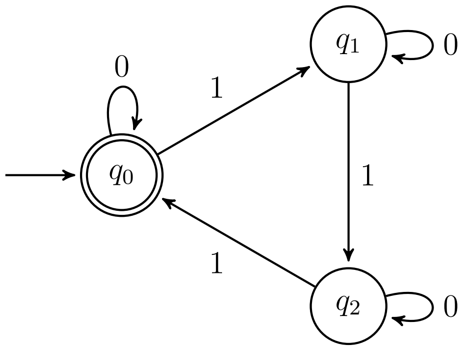
(Again, we omit many arrows to simplify the figure.)

Example 3–1: Let A and B be languages over $\Sigma = \{0, 1\}$:

$$\begin{aligned} A &= \{w \mid \text{Number of 1's in } w \text{ is a multiple of } 2.\} \\ B &= \{w \mid \text{Number of 1's in } w \text{ is a multiple of } 3.\} \end{aligned}$$

The following DFAs recognize these languages:





Give the state diagram of

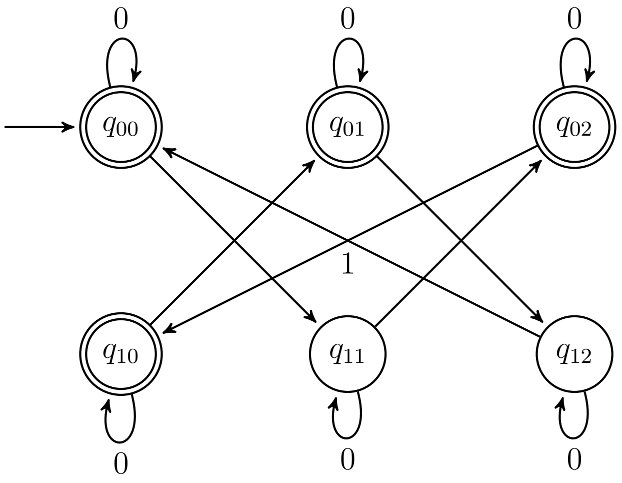
a) a DFA

b) an NFA

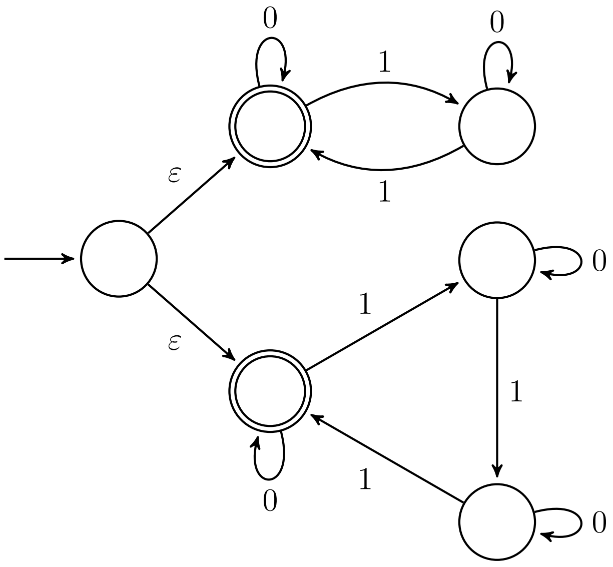
that recognizes the language $A \cup B$.

Use the machines given above in your construction.

Solution: Using the given procedure we obtain the following DFA: (All the arrows in the middle are labeled 1.)



Even in this very simple example, the resulting machine is kind of complicated. The NFA alternative is:



Theorem: The class of regular languages is closed under the concatenation operation. In other words, if A and B are regular languages, so is $A \circ B$.

Sketch of Proof: Given two strings $v \in A$ and $w \in B$ we can easily form the concatenated string vw. But the opposite problem is more difficult.

Given a string, how can we divide it into two parts such that the first part belongs to the first language and the second part to the second?

Perhaps we can give the string to the machine of A. If it enters an accept state during the computation, we can stop that machine and give the rest of the string to the machine of B.

Unfortunately, this trick may not always work. For example, let

$$\begin{aligned} A &= \{w \mid w \text{ ends with } 11.\} \\ B &= \{w \mid w \text{ contains odd number of } 1\text{'s.}\} \end{aligned}$$

What is the correct way to separate the string

$$1011010011001$$

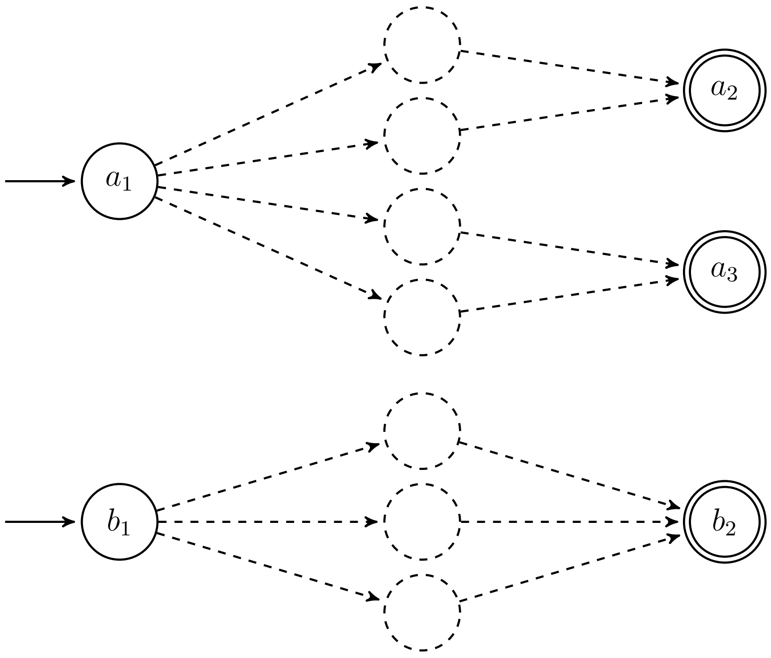
into two parts?

$$\begin{aligned} &1011 - 010011001 \\ \text{or}\quad &1011010011 - 001 \quad ? \end{aligned}$$

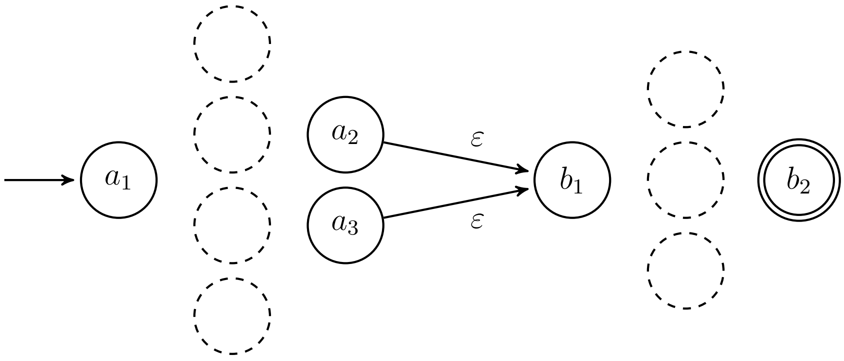
We have to think in more general terms. Rather than proceeding to the machine of B when the machine of A enters an accept state *for the first time*, we have to proceed to the second part *whenever* the machine of A enters an accept state.

This is very easy using NFAs and $\varepsilon-$transitions.

Let the machines recognizing the languages A and B be given as:



- Connect all accept states of the first machine to the start state of the second machine.
- These accept states are no longer accept states.

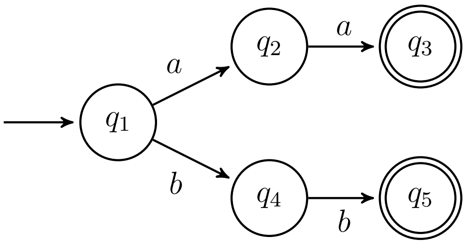


Example 3–2: Let A and B be languages over $\Sigma = \{a, b, c\}$:

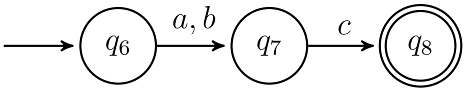
$$A = \{aa, bb\}, \qquad B = \{ac, bc\}$$

Find NFAs recognizing A, B and $A \circ B$.

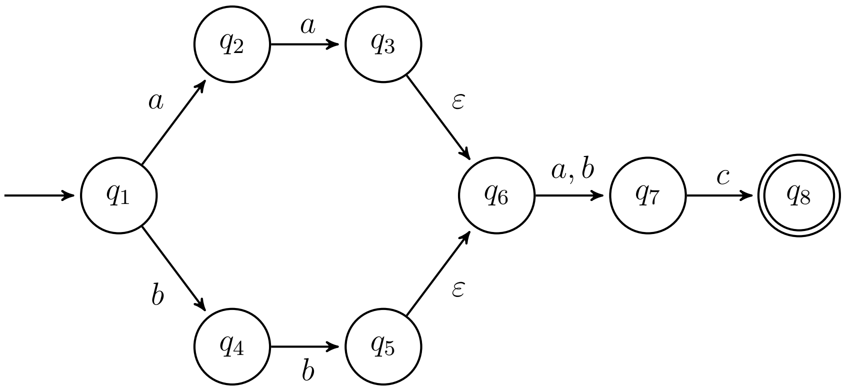
Solution: A:



B:



$A \circ B$:



Check that this machine recognizes the language:

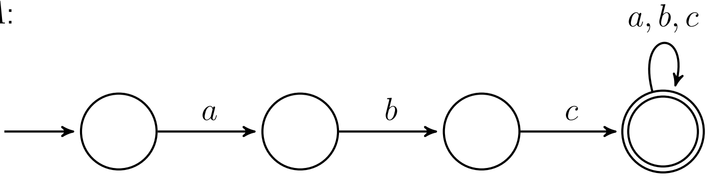
$$\{aaac, aabc, bbac, bbbc\}$$

Example 3–3: Let A and B be languages over $\Sigma = \{a, b, c\}$:

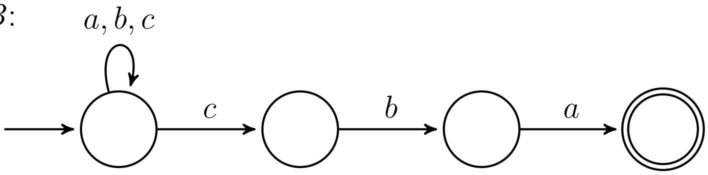
$$\begin{aligned} A &= \{w \mid w \text{ starts with } abc.\} \\ B &= \{w \mid w \text{ ends with } cba.\} \end{aligned}$$

Find NFAs recognizing A, B, and $A \circ B$.

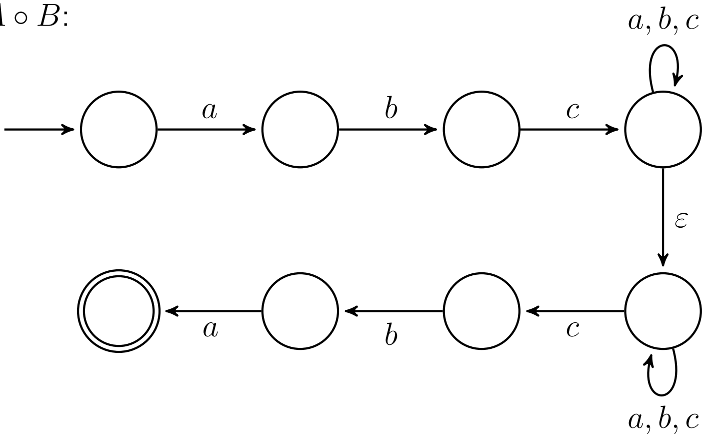
Solution: A:



B:



$A \circ B$:

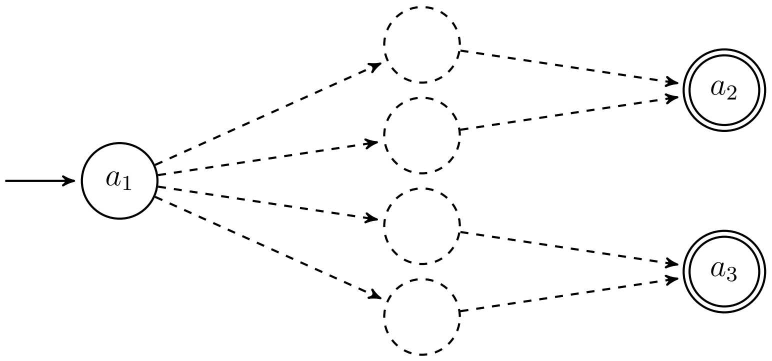


Note that in both this exercise and the previous one, the machines could be simplified by eliminating some states.

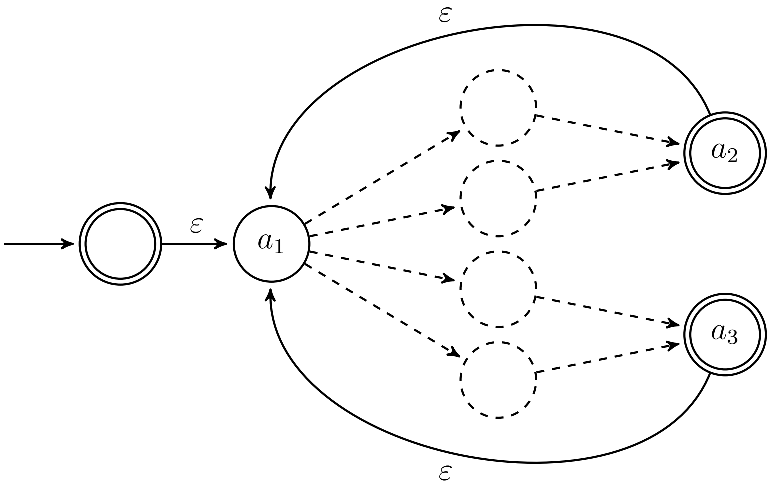
Theorem: The class of regular languages is closed under the star operation.

Sketch of Proof: This is similar to concatenation. We have to connect the accept states to start state.

Suppose the following machine recognizes A:



Then the following machine recognizes A^*:



We have to be careful about the start state. The new machine must accept ε because $\varepsilon \in A^*$ by definition.

Suppose the start state is not accepting. Let's make it accepting in the new machine. This solves our problem of ε but the machine may accept some strings that are not part of the language.

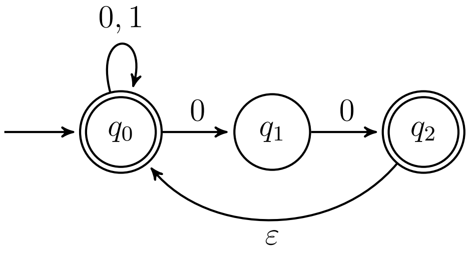
To prevent this problem, we need an extra state which is both starting and accepting state.

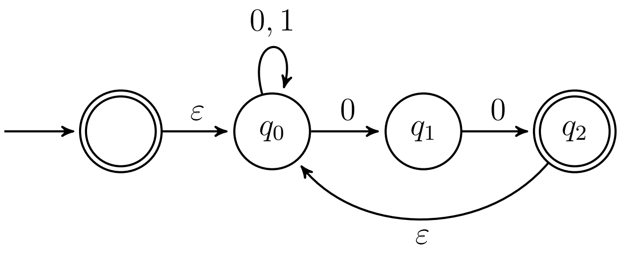
Example 3–4: Let A be a languages over $\Sigma = \{0, 1\}$:

$$A = \{w \mid w \text{ ends with } 00.\}$$

Find an NFA that recognizes A^*.

Solution: You should see that the first machine does not work as intended, but the second one does:





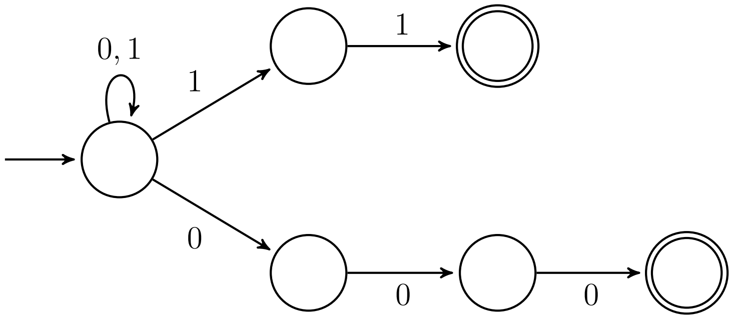
Example 3–5: Let A be a language over $\Sigma = \{0, 1\}$.

$$A = \{w \mid w \text{ ends with } 11 \text{ or } 000.\}$$

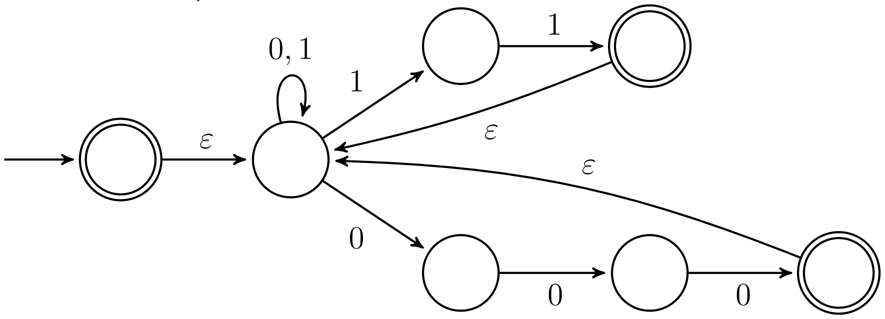
a) Find an NFA that recognizes A.

b) Find an NFA that recognizes A^*.

Solution: a)



b)

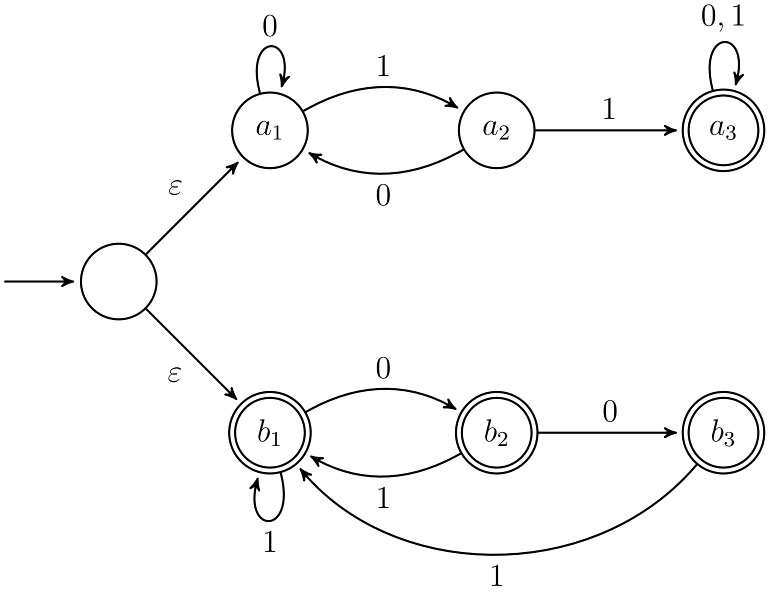


Example 3–6: Let A and B be languages over $\Sigma = \{0, 1\}$.

$$A = \{w \mid w \text{ contains } 11.\}$$
$$B = \{w \mid w \text{ does not contain } 000.\}$$

Give the state diagram of an NFA that recognizes $A \cup B$.

Solution:



Note that if this were a DFA, 0 after b_3 would result in a trap state. But we do not need to show it in an NFA diagram.

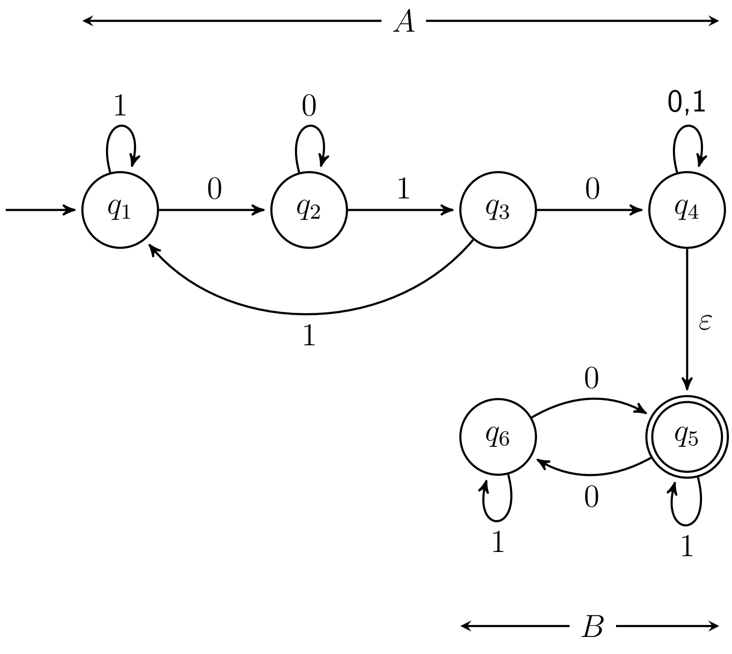
Example 3–7: Let A and B be languages over $\Sigma = \{0, 1\}$.

$$A = \Big\{ w \;\Big|\; w \text{ contains } 010. \Big\}$$

$$B = \Big\{ w \;\Big|\; w \text{ contains even number of zeros.} \Big\}$$

Give the state diagram of an NFA that recognizes $A \circ B$.

Solution:



EXERCISES

3–1) The languages A and B are defined over $\Sigma = \{0,1\}$ as:

$$\begin{aligned} A &= \big\{w \,\big|\, w \text{ starts with } 00.\big\} \\ B &= \big\{w \,\big|\, w \text{ starts with } 11.\big\} \end{aligned}$$

Design a DFA and an NFA that recognizes the language $A \cup B$.

3–2) The languages A and B are defined over $\Sigma = \{0,1\}$ as:

$$\begin{aligned} A &= \big\{w \,\big|\, w \text{ ends with } 00.\big\} \\ B &= \big\{w \,\big|\, w \text{ contains at least two } 1\text{'s.}\big\} \end{aligned}$$

Design an NFA that recognizes the language $A \circ B$.

3–3) The language A is defined over $\Sigma = \{0,1\}$ as:

$$A = \big\{0110, 110\big\}$$

Design an NFA that recognizes the language A^*.

3–4) The language A is defined over $\Sigma = \{0,1\}$ as:

$$A = \big\{w \,\big|\, w \text{ starts and ends with } 1 \text{ and } |w| = 3.\big\}$$

Design an NFA that recognizes the language A^*.

3–5) The languages A, B and C are defined over $\Sigma = \{a, b, c\}$ as:

$$\begin{aligned} A &= \{w \mid w \text{ starts with } a.\} \\ B &= \{w \mid w \text{ ends with } b.\} \\ C &= \{w \mid w \text{ contains } cc.\} \end{aligned}$$

Design an NFA that recognizes the language $\big(A \circ B \cup C\big)^*$.

3–6) The languages A, B and C are defined over $\Sigma = \{a, b\}$ as:

$$\begin{aligned} A &= \{w \mid w = (ab)^n aa, \quad n \geqslant 0.\} \\ B &= \{w \mid w \text{ contains even number of symbols.}\} \\ C &= \{w \mid w \text{ starts and ends with } b \text{ and } |w| \geqslant 3.\} \end{aligned}$$

Design an NFA that recognizes the language $A \circ B \cup C$.

3–7) Let A be a regular language. Is $\overline{A}$ (set complement of A) also a regular language?

3–8) Let A and B be regular languages. Is $A \cap B$ also a regular language?

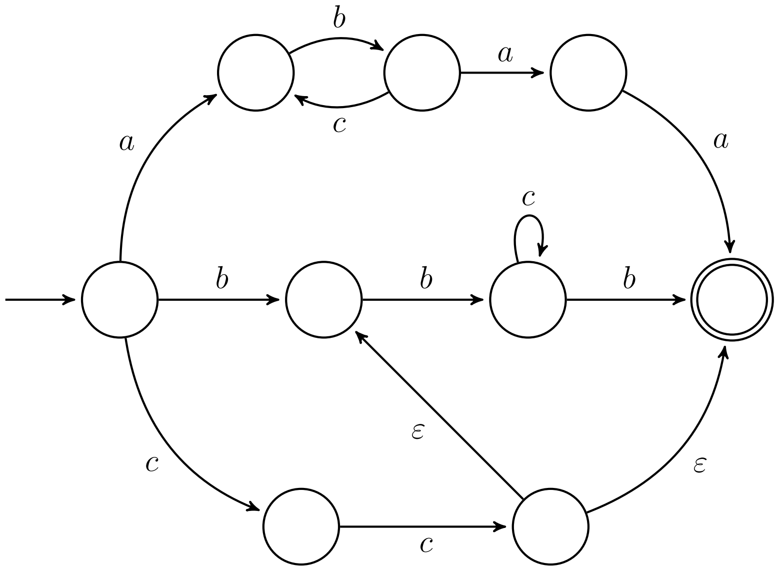
3–9) Let A and B be regular languages. Is $A \setminus B$ (set difference) also a regular language?

3–10) Let A be a regular language. Let's obtain the language B from A by reversing all strings. For example,

$$bbcb \in A \quad \Rightarrow \quad bcbb \in B$$

We denote this as $B = A^R$. Is B also a regular language?

3–11) Consider the following NFA:

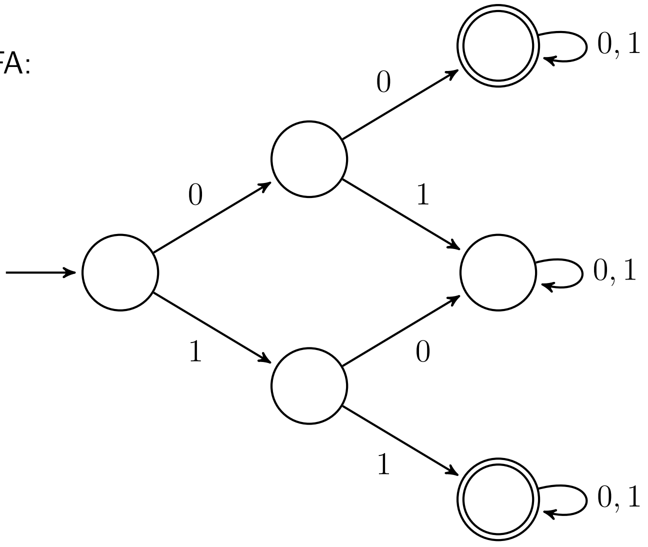


Let the language recognized by this NFA be A. The language defined as $B = A^R$ must be regular by the previous exercise.

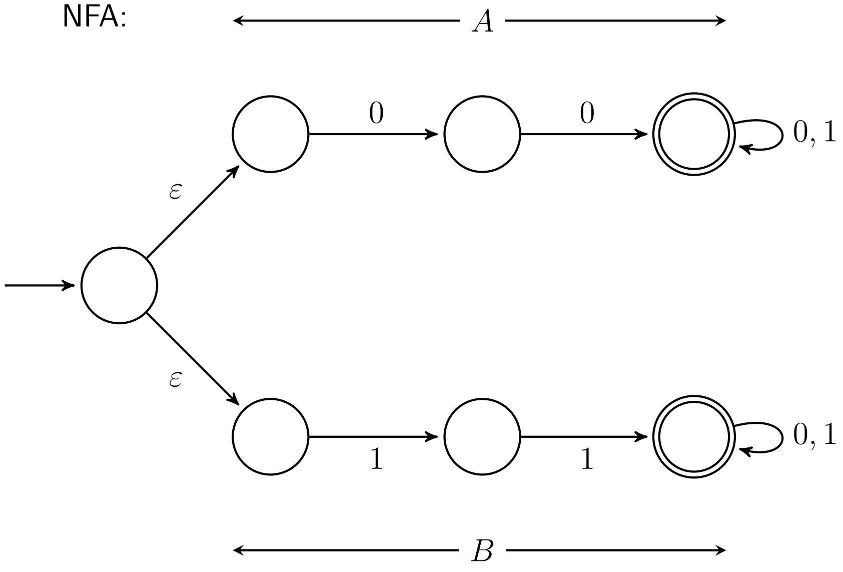
Find an NFA for it.

ANSWERS

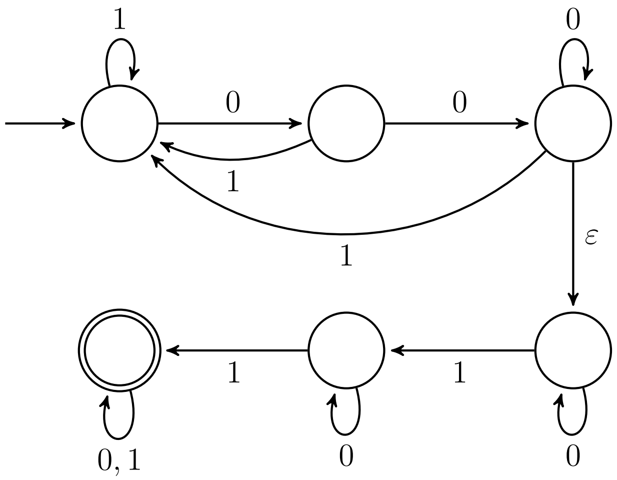
3–1) DFA:



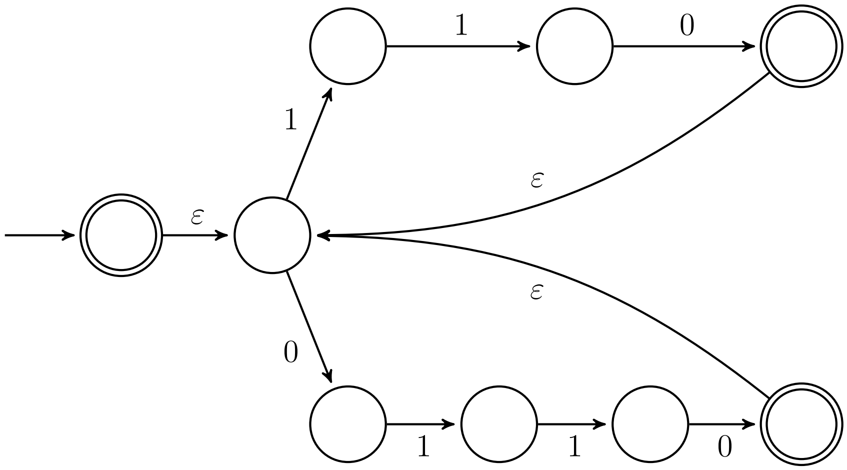
NFA:



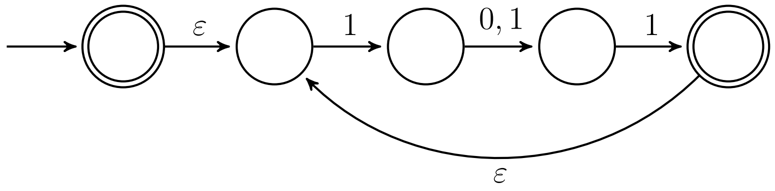
3–2)



3–3)

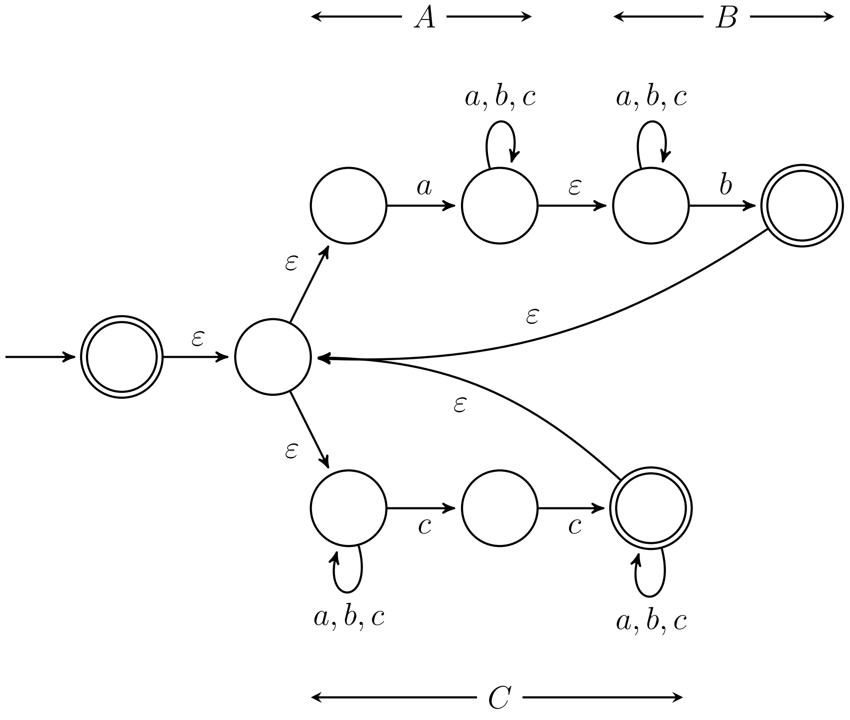


3–4)



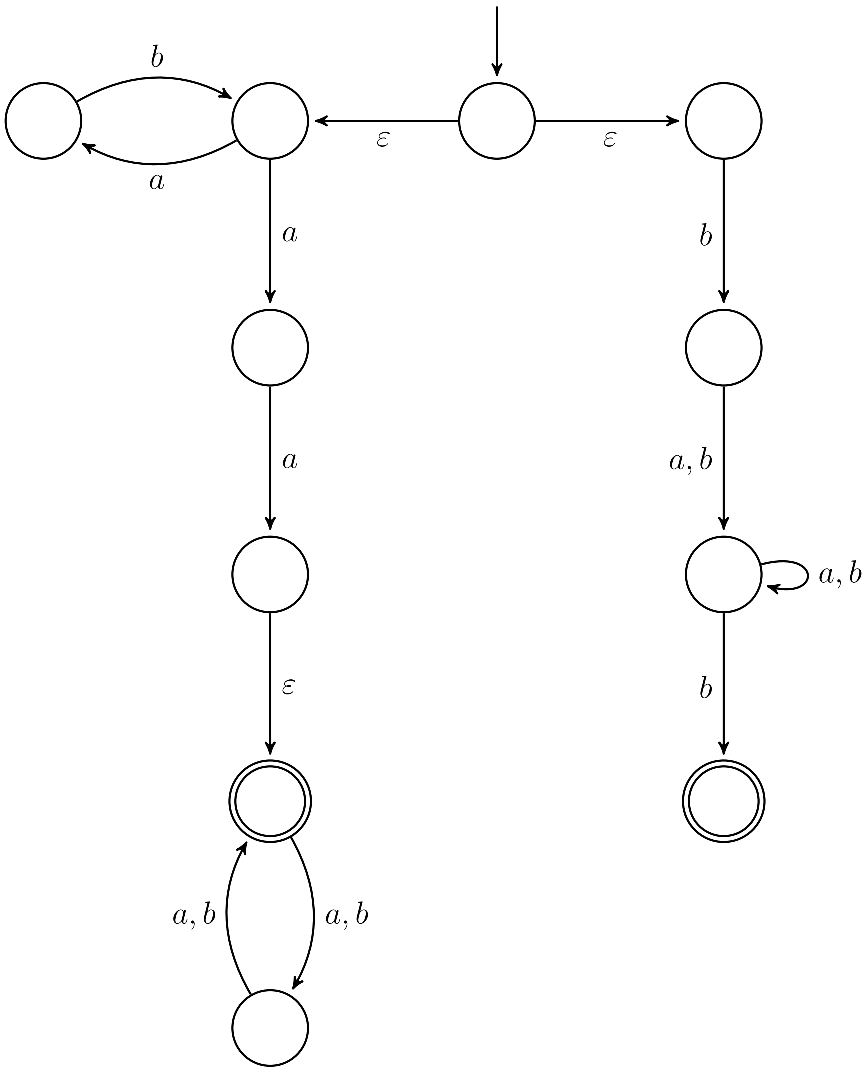
ε
1
$0, 1$
1
ε

3–5)



A
B
a, b, c
a, b, c
a
ε
b
ε
ε
ε
ε
ε
c
c
a, b, c
a, b, c
C

3–6)



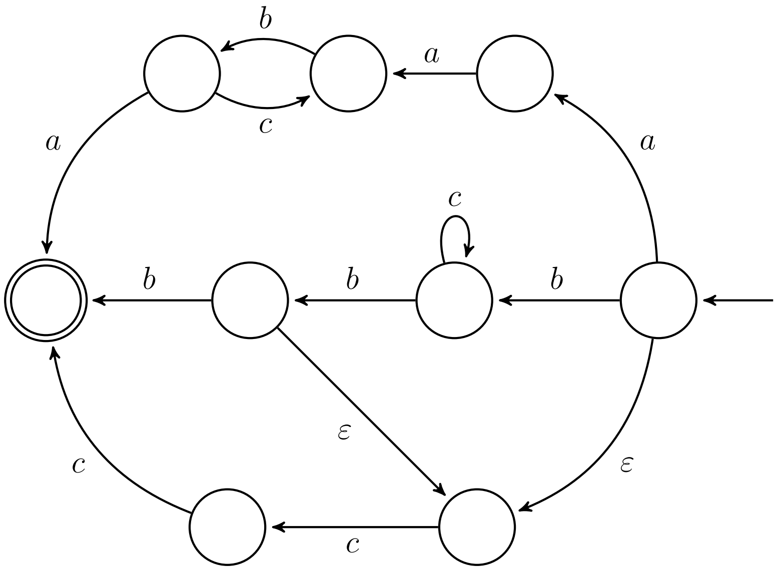
3–7) Yes. Given a DFA for A, we can make all accepting states non-accepting and vice versa.

3–8) Yes. $\overline{(\overline{A} \cup \overline{B})} = A \cap B$

3–9) Yes. $A \setminus B = A \cap \overline{B}$.

3–10) Yes. Given a DFA for A, we can reverse arrows and switch accept and start states to obtain a DFA for B. (What if there are more than one accept states?)

3–11) Yes. We can find an NFA for language B by reversing arrows:



Week 4

Regular Expressions

4.1 Definition

Regular expressions are a way to represent a set of strings (i.e. a language) in an alphabet. We will use the following operations to combine strings and obtain new ones:

- Union: $\cup$
- Concatenation: $\circ$
- Star: *

These are the same operations we used on languages in the last chapter. Although a string and a set of strings are different concepts, we will use the same notation, because we will think of regular expressions as shorthand notations expressing languages. For example,

$$00 \cup 10 \quad \text{and} \quad \{00\} \cup \{10\}$$

or

$$\{001, 111\}^* \quad \text{and} \quad (001 \cup 111)^*$$

will mean the same thing. (We omit the concatenation operator.)

Formal Definition: We say that R is a **regular expression** if R is

1. $\varnothing$.
2. ε.
3. a where $a \in \Sigma$.
4. $X \cup Y$ where X and Y are regular expressions.
5. $X \circ Y$ where X and Y are regular expressions.
6. X^* where X is a regular expression.

Operation precedence is star first, concatenation second, and union last. For example,

$$ab^* \cup c$$

is equivalent to

$$\big(a(b^*)\big) \cup c$$

This is different from

$$a\big(b^* \cup c\big) \quad \text{or} \quad \big(ab\big)^* \cup c$$

Another important notation is $^+$. It is like * but it means at least one symbol is included.

$$a^+ = \{a, aa, aaa, \ldots\} = aa^*$$

Note that $\varnothing$ and ε are different things. There is no string in the language $\varnothing$ but there is one string in the language ε. It is the empty string. For example,

$$00 \cup \varnothing = \{00\} \quad \text{but} \quad 00 \cup \varepsilon = \{00, \varepsilon\}$$

$$(00 \cup \varnothing)11 = \{0011\} \quad \text{but} \quad (00 \cup \varepsilon)11 = \{0011, 11\}$$

Example 4–1: Verbally describe the languages represented by the following regular expressions. Assume $\Sigma = \{0, 1\}$.

a) $1(0 \cup 1)^*$

b) $1\Sigma^*$

c) $\Sigma^* 1 \Sigma^*$

d) $0^*(\varepsilon \cup 1)0^*$

e) $0^* 1 0^*$

f) $(0 \cup 1)(0 \cup 1)(0 \cup 1)$

g) $\Sigma\Sigma\Sigma$

h) $(\Sigma\Sigma\Sigma)^*$

i) $(\Sigma\Sigma\Sigma)^+$

j) $\varnothing^*$

Solution: a) The strings that start with 1.

b) Same as a).

c) Contains at least one 1.

d) Contains at most one 1.

e) Contains exactly one 1.

f) String length exactly three.

g) Same as f).

h) String length a multiple of three.

i) String length a positive multiple of three.

j) ε

Example 4–2: Using the following verbal descriptions of languages, find the corresponding regular expressions. Assume $\Sigma = \{a, b, c\}$.

a) Strings of length at least 6 that start and end with aaa.

b) Strings of length at least 7 that start and end with aaa.

c) Strings of length at least 3 that start and end with aaa.

d) Strings where there is no a after any b and there is no c before any b.

e) Strings that do not contain bb.

f) Strings that start and end with the same symbol.

g) Strings that contain an even number of c's.

h) Strings of length exactly 2.

i) Strings of length less than or equal to 2.

j) Strings that contain at least two c's.

Solution: a) $aaa\Sigma^*aaa$

b) $aaa\Sigma^+aaa$

c) $aaa\Sigma^*aaa \cup aaaaa \cup aaaa \cup aaa$

d) $a^*b^*c^*$

e) $(b \cup \varepsilon)\left[(a \cup c)^+b\right]^*(a \cup c)^*$

f) $a\Sigma^*a \cup b\Sigma^*b \cup c\Sigma^*c \cup a \cup b \cup c$

g) $\left[(a \cup b)^*c(a \cup b)^*c(a \cup b)^*\right]^* \cup (a \cup b)^*$

h) $(a \cup b \cup c)(a \cup b \cup c)$

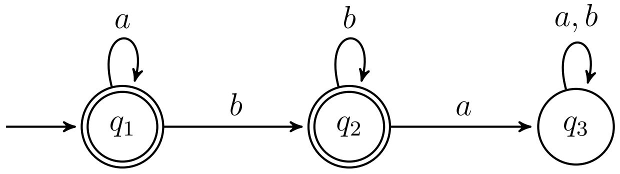
i) $(a \cup b \cup c \cup \varepsilon)(a \cup b \cup c \cup \varepsilon)$

j) $\Sigma^*c\Sigma^*c\Sigma^*$

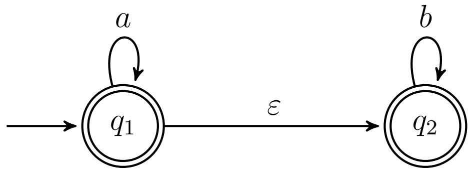
Example 4–3: Give the state diagram of a DFA or an NFA that recognizes the language A over alphabet $\Sigma = \{a, b\}$ where

$$A = \{w \mid w = a^* b^*\}$$

Solution: DFA:

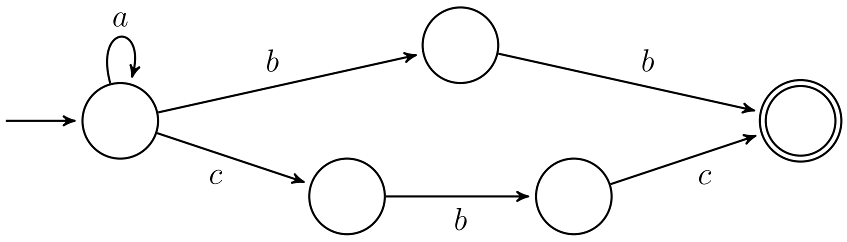


NFA:



As usual, NFA is simpler and reflects the structure of the regular expression better.

Example 4–4: Find a regular expression corresponding to the language of the following NFA over the alphabet $\Sigma = \{a, b, c\}$:



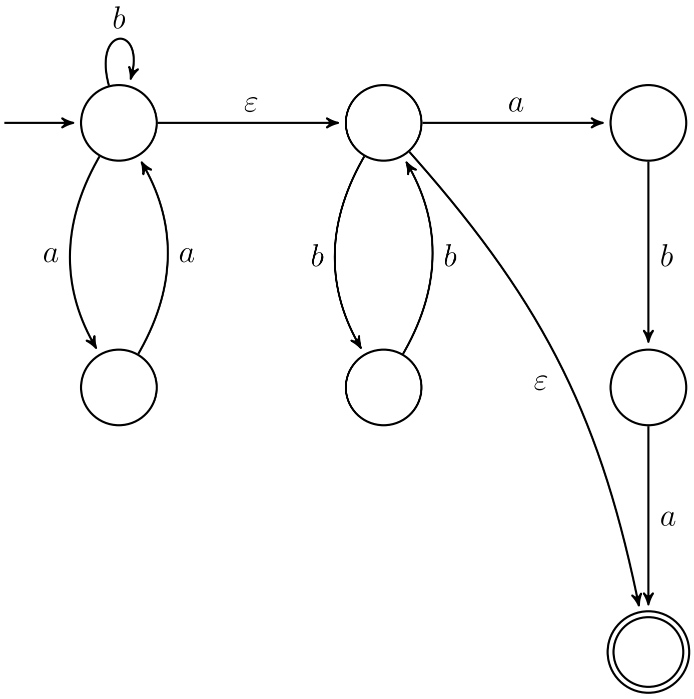
Solution: $a^*(bb \cup cbc)$

Example 4–5: Consider the regular expression

$$(b \cup aa)^* \, (bb)^* \, (\varepsilon \cup aba)$$

Find an NFA recognizing a language equivalent to this.

Solution: We have to think in terms of three stages. Union means parallel and concatenation means series connections:



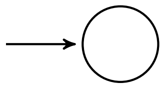
Based on those examples, it seems that there is a regular expression for a given NFA and there is an NFA for a given regular expression. But is that correct in general?

4.2 Regular Languages and Regular Expressions

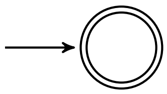
Theorem: A language is regular if and only if some regular expression describes it.

Proof - Part 1: Given a regular expression, find an NFA that recognizes it.

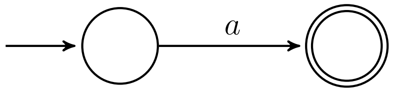
1. $R = \varnothing$:



2. $R = \varepsilon$:



3. $R = a$:



The cases for union, concatenation, and star operations were covered in the previous chapter, but very briefly:

4. $X \cup Y$: Connect in parallel with $\varepsilon-$arrows.

5. $X \circ Y$: Connect in series with $\varepsilon-$arrows. (Accept states of X to start state of Y.)

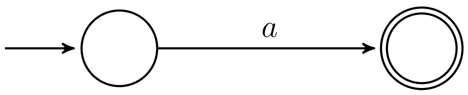
6. X^*: Connect accept states to start state with $\varepsilon-$arrows. (Add an extra start state.)

Example 4–6: Convert the given regular expression into an NFA.

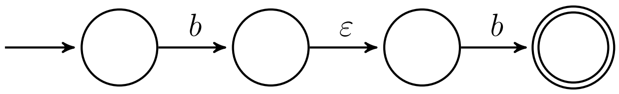
$$(a \cup bb)^*$$

Solution:

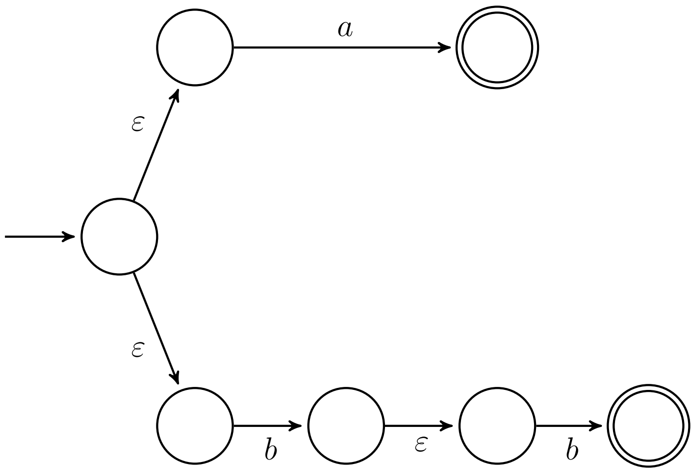
a :



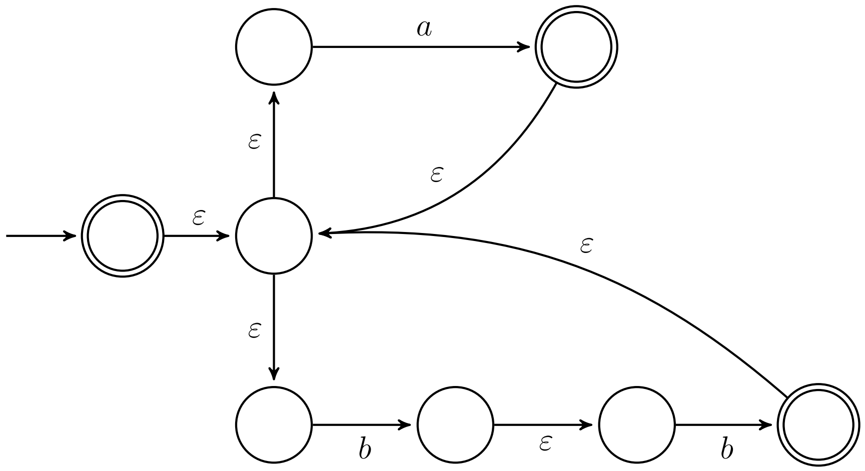
bb :



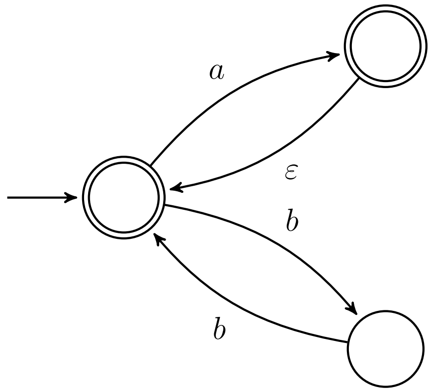
$a \cup bb$:



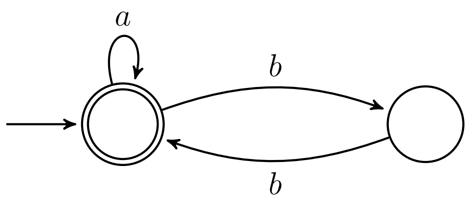
$(a \cup bb)^*$:



This procedure gives an automaton that works, but it is not the simplest one. An alternative solution is:



An even simpler one is:

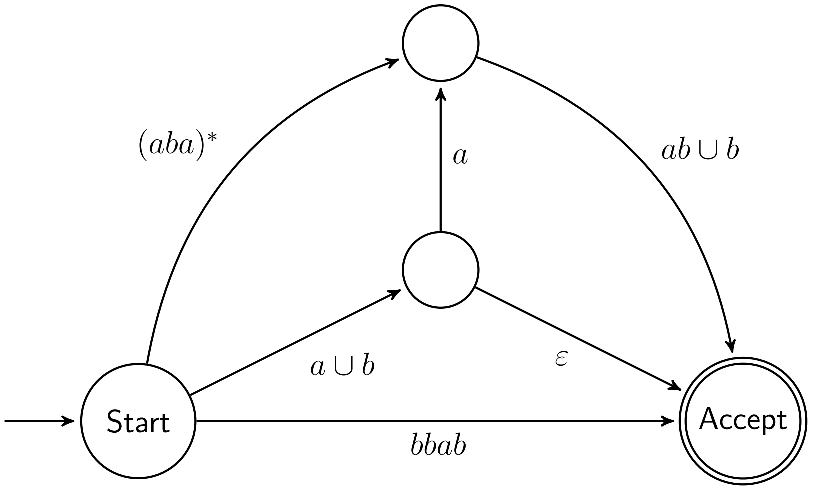


Proof - Part 2: Given an NFA, find a regular expression equivalent to its language.

For this part, we will slightly modify the definition of an NFA. We will consider machines where:

- There is a single accept state, distinct from the start state.
- Arrows do not enter the start state.
- Arrows do not leave the accept state.
- Arrow labels are regular expressions.

For example, this is such an automaton:



Given an NFA, we can put it into that form by adding new start and accept states if necessary. (We should connect them to old ones by $\varepsilon-$transitions.)

The result is simpler than a usual NFA in some sense.

Then, we will eliminate all intermediate states one by one until we obtain a $2-$state machine. The label on the last remaining arrow will give us the regular expression we are looking for.

Here, the rule is that the language of the machine must stay the same as we eliminate states. Therefore we should modify arrow labels as follows:

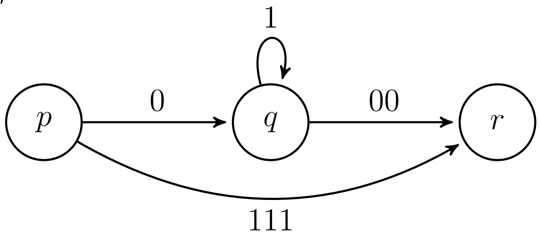
Suppose p, q, r are states, W, X, Y, Z are regular expressions such that:

- Transition $p \to r$ has label W.
- Transition $p \to q$ has label X.
- Transition $q \to q$ has label Y.
- Transition $q \to r$ has label Z.

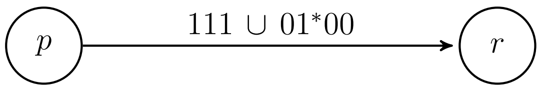
In that case, we can remove the state q and relabel the transition $p \to r$ as:

$$W \cup XY^*Z$$

For example,

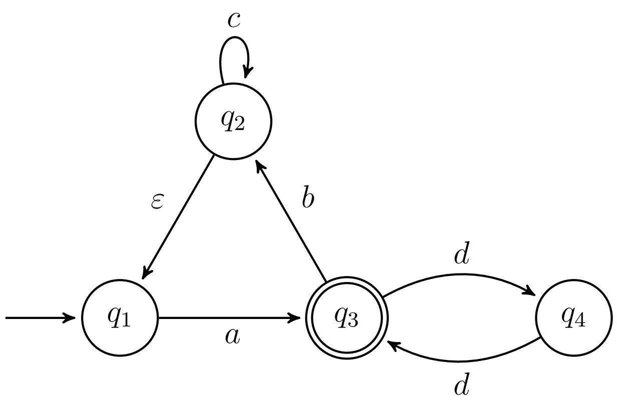


will be:

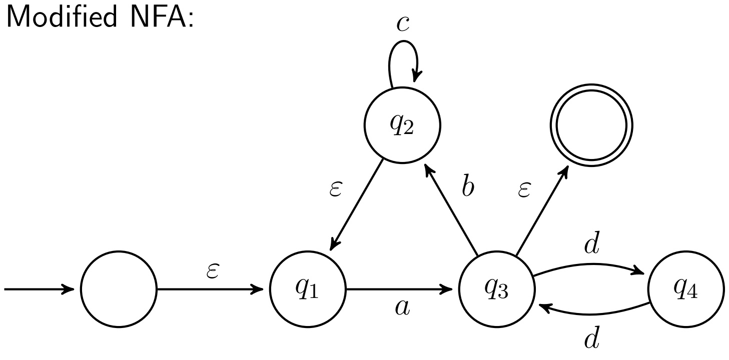


Of course, this procedure should be followed for any pair of states that have q as an intermediate state.

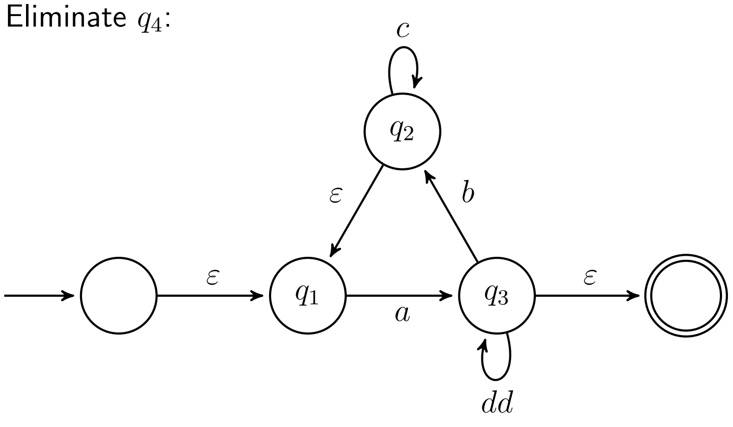
Example 4–7: Find the regular expression equivalent to the following NFA:



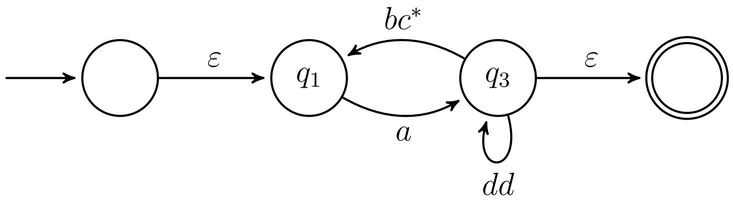
Solution: Modified NFA:



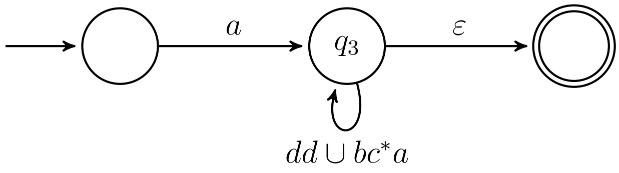
Eliminate q_4:



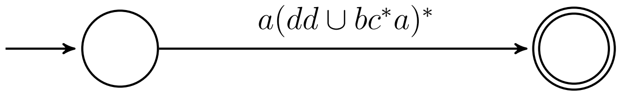
Eliminate q_2:



Eliminate q_1:



Finally:



So the answer is:

$$a(dd \cup bc^*a)^*$$

Alternative answers are:

$$[a(dd)^*bc^*]^*a(dd)^*$$

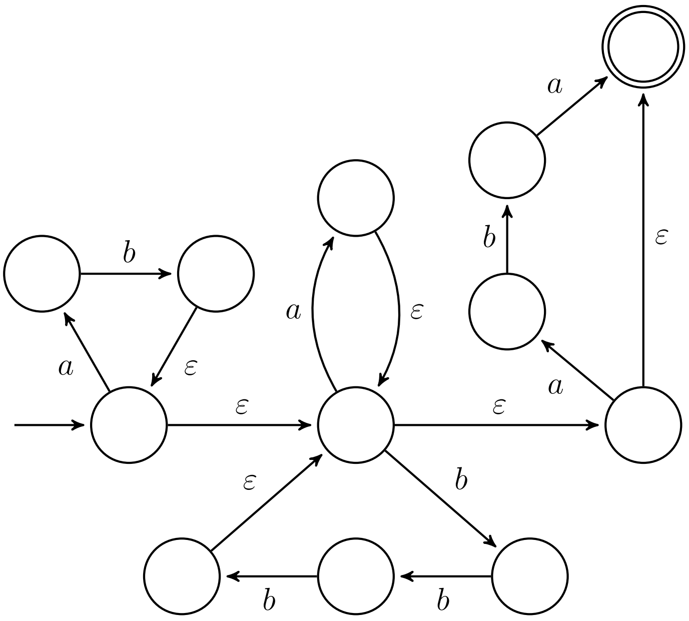
or

$$a[(dd)^*(bc^*a)^*]^*$$

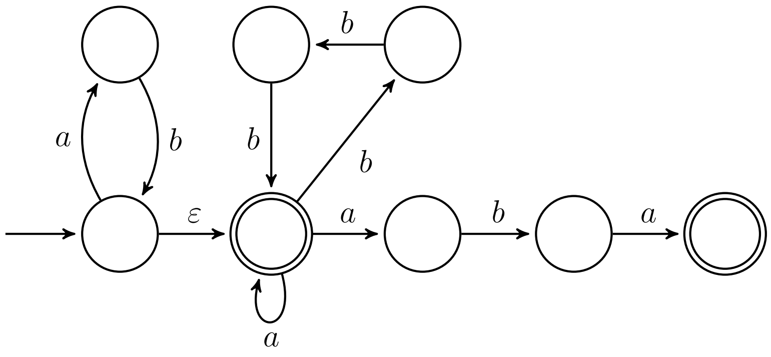
Example 4–8: Find an NFA recognizing the regular expression:

$$(ab)^* \left(a^* \cup bbb\right)^* \left(aba \cup \varepsilon\right)$$

Solution:



OR, this can be simplified as:



EXERCISES

4–1) The language L is described by the regular expression

$$1(01)^* \cup 0(11)^*$$

over $\Sigma = \{0, 1\}$. Are the following strings in L?

a) 10101

b) 0

c) 101010

d) 0111

4–2) The language L is described by the regular expression

$$(\Sigma^* a \Sigma^* a \cup bb)b^*$$

over $\Sigma = \{a, b\}$. Are the following strings in L?

a) $aabbbbb$

b) $bbbba$

c) $bbababab$

d) a

Verbally describe the following languages defined over the alphabet $\Sigma = \{0, 1\}$:

4–3) $0(10)^* \cup 0(10)^*1 \cup 1(01)^* \cup 1(01)^*0$

4–4) $0(\Sigma\Sigma)^*\Sigma \cup 1\big(\Sigma\Sigma\big)^*$

4–5) $(0 \cup 1)^*\, 0(0 \cup 1)^*\, 1(0 \cup 1)^* \cup (0 \cup 1)^*\, 1(0 \cup 1)^*\, 0(0 \cup 1)^*$

Find regular expressions for the following languages defined over the alphabet $\Sigma = \{a, b, c\}$:

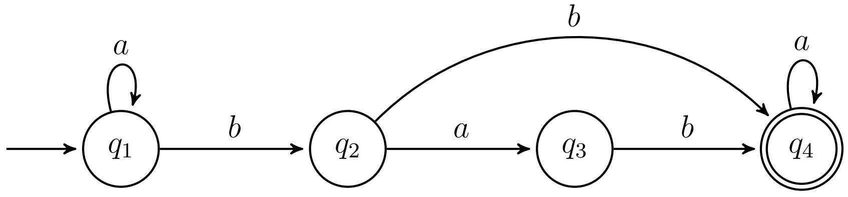
4–6) Strings of odd length; string length is three or greater; first and last symbols are c.

4–7) Strings of length at least two. Strings start with a or b and end with c. Strings do not contain ab or ba.

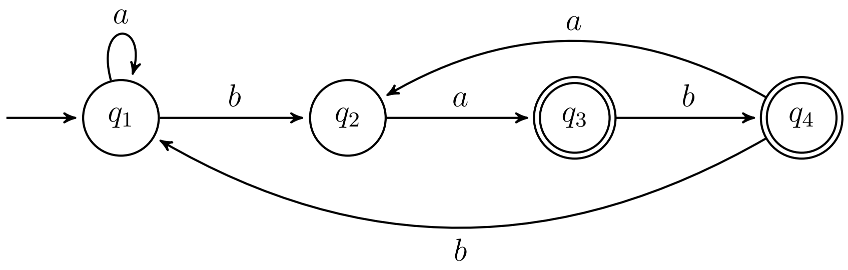
4–8) Strings contain at most two c's.

Find a regular expression equivalent to the language recognized by the following NFAs:

4–9)

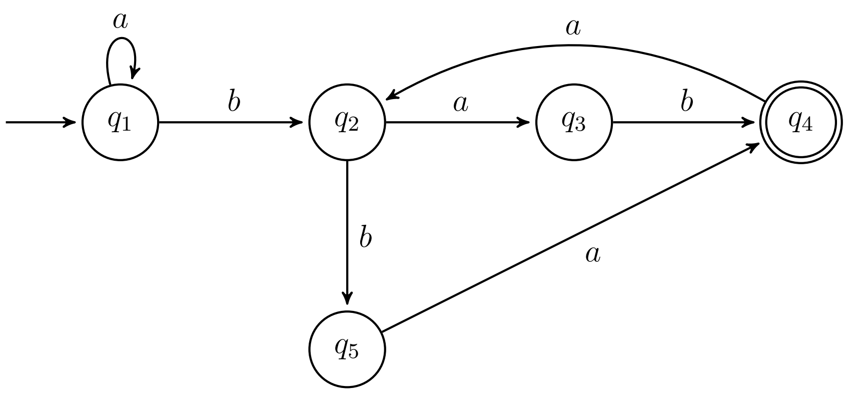


4–10)

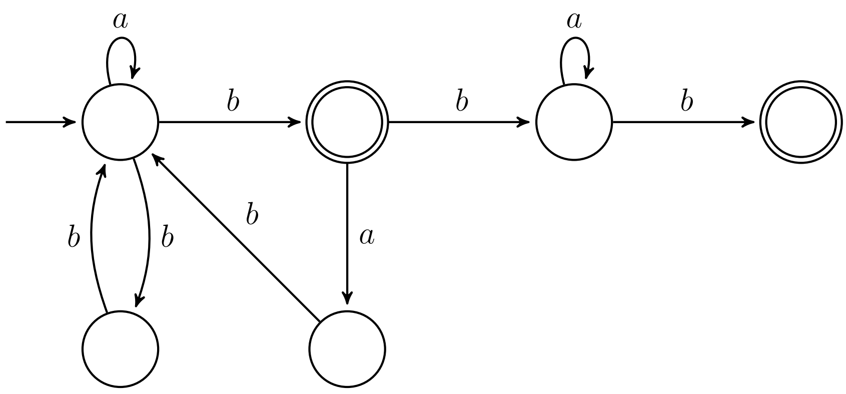


Find a regular expression equivalent to the language recognized by the following NFAs:

4–11)

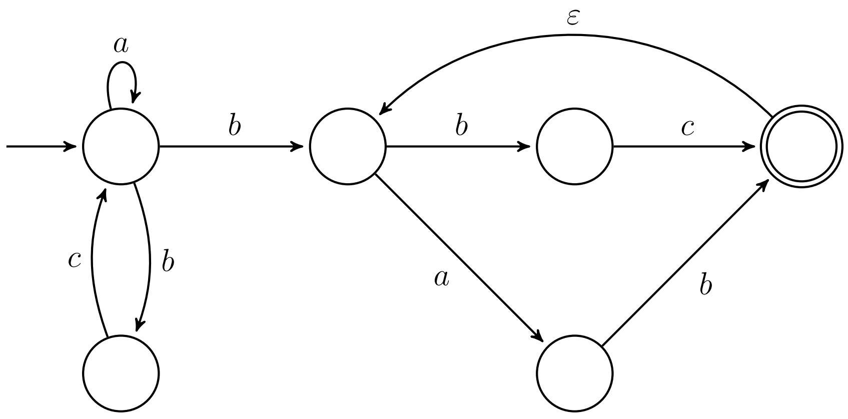


4–12)

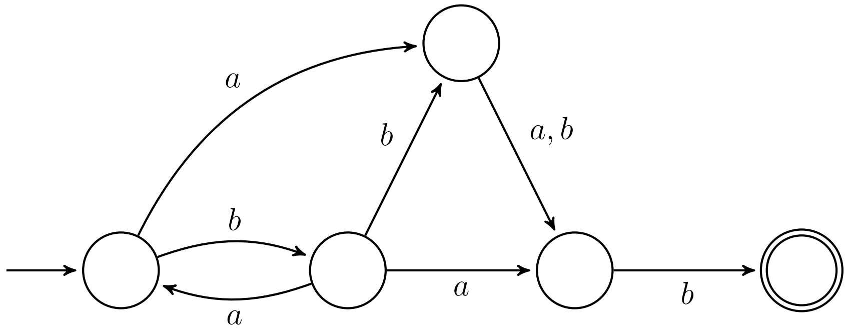


Find a regular expression equivalent to the language recognized by the following NFAs:

4–13)

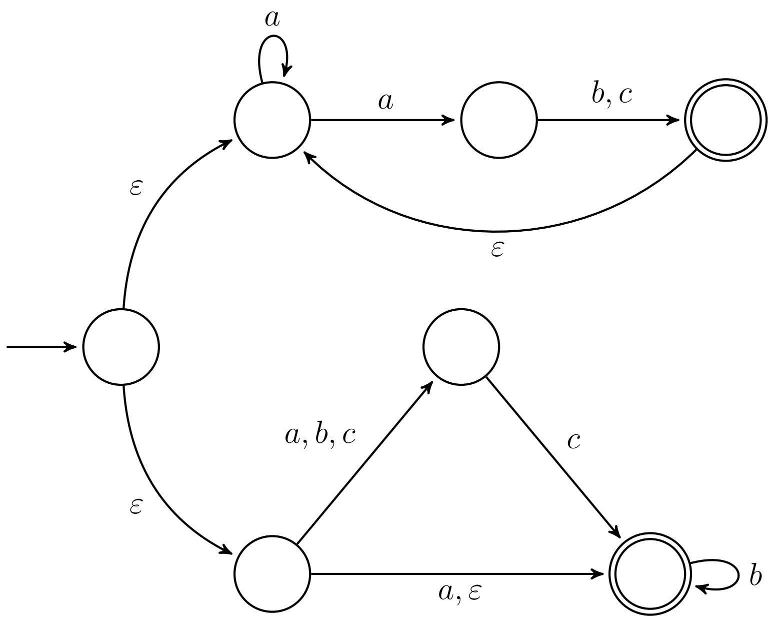


4–14)

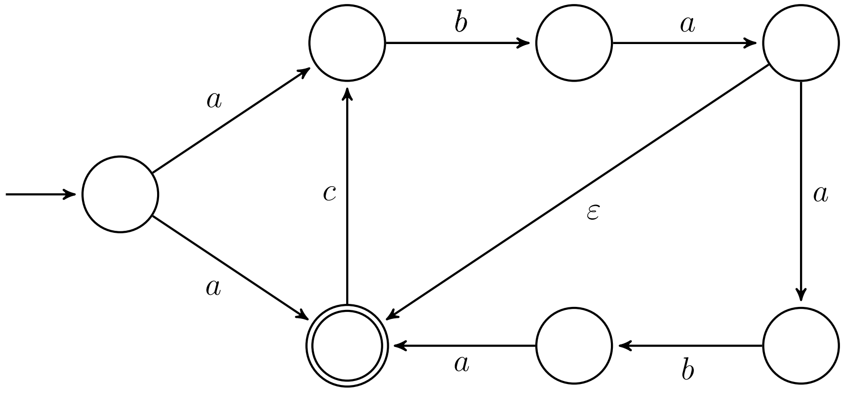


Find a regular expression equivalent to the language recognized by the following NFAs:

4–15)



4–16)



Find an NFA that recognizes the language given by the following regular expression:

4–17) $(\Sigma^* a \Sigma^* a \cup bb)b^*$

4–18) $(\varepsilon \cup 000 \cup 11)(01)^*$

4–19) $(aaa \cup aba)^* bab^*$

4–20) $ab^* \left[ba^* \cup (ba)^*\right]$

4–21) $(a \cup aba \cup ba)^*$

4–22) $\left[(bb \cup aba)^* baa\right]^*$

ANSWERS

4–1)

a) Yes.

b) Yes.

c) No.

d) No.

4–2)

a) Yes.

b) No.

c) Yes.

d) No.

4–3) Alternating sequences of 0's and 1's, or
Strings that do not contain 00 or 11, of length one or longer.

4–4) String length is at least one. If a string starts with 0, it is of even length. If it starts with 1, it is of odd length.

4–5) Strings contain at least one 0 and at least one 1.

4–6) $c\Sigma\big(\Sigma\Sigma\big)^*c$

4–7) $\big(a^+c^+ \cup b^+c^+\big)^+$

4–8) $(a \cup b)^*(c \cup \varepsilon)(a \cup b)^*(c \cup \varepsilon)(a \cup b)^*$

4–9) $a^*b(b \cup ab)a^*$

4–10) $\left[a \cup b(aba)^*abb\right]^* b\big(aba\big)^* \big(a \cup ab\big)$

4–11) $a^*b\big(aba \cup baa\big)^* \big(ab \cup ba\big)$

4–12) $(a \cup bb \cup bab)^*\, b\, (\varepsilon \cup ba^*b)$

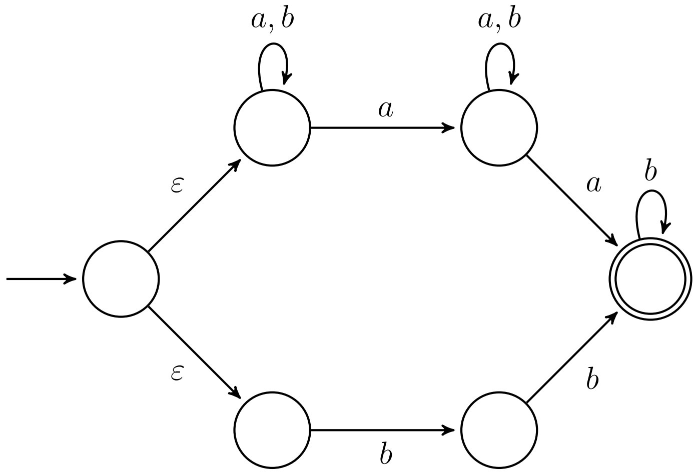
4–13) $\big(a \cup bc\big)^*\, b\, \big(bc \cup ab\big)^+$

4–14) $\big(ba\big)^* \Big[ba \cup \big(a \cup bb\big)\big(a \cup b\big)\Big] b$

4–15) $\Big[a^*\big(ab \cup ac\big)\Big]^+ \cup \big(\Sigma c \cup a \cup \varepsilon\big) b^*$

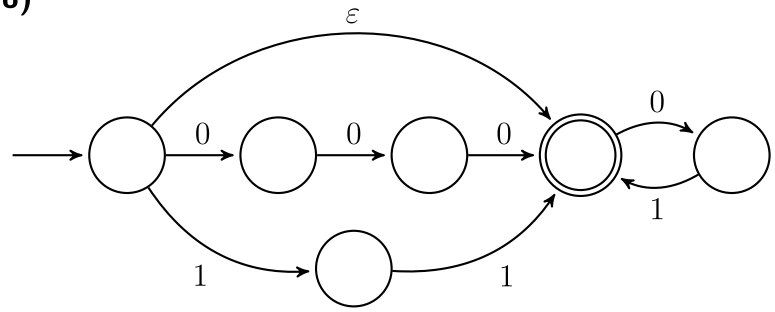
4–16) $\big(a \cup aba \cup abaaba\big)\big(cba \cup cbaaba\big)^*$

4–17)



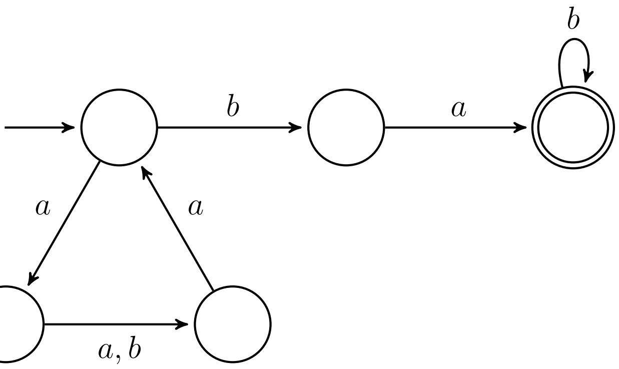
a, b
a, b
a
ε
a
b
ε
b
b

4–18)



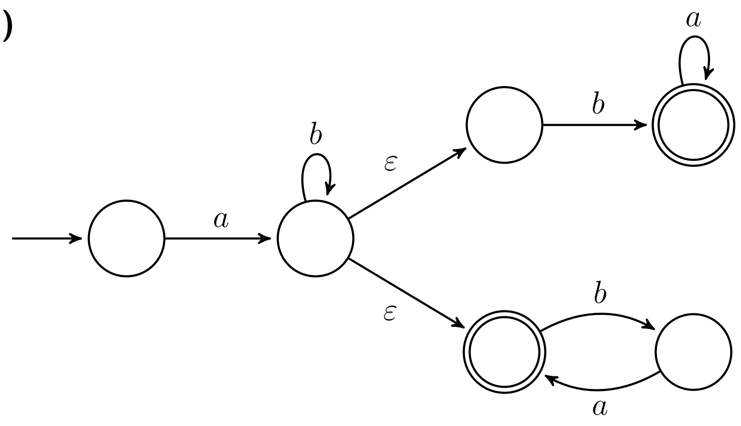
ε
0
0
0
0
1
1
1

4–19)

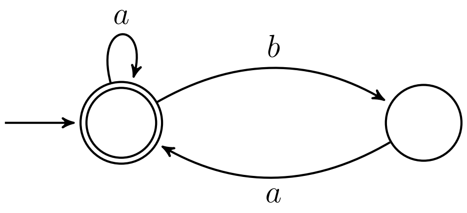


b
b
a
a
a
a, b

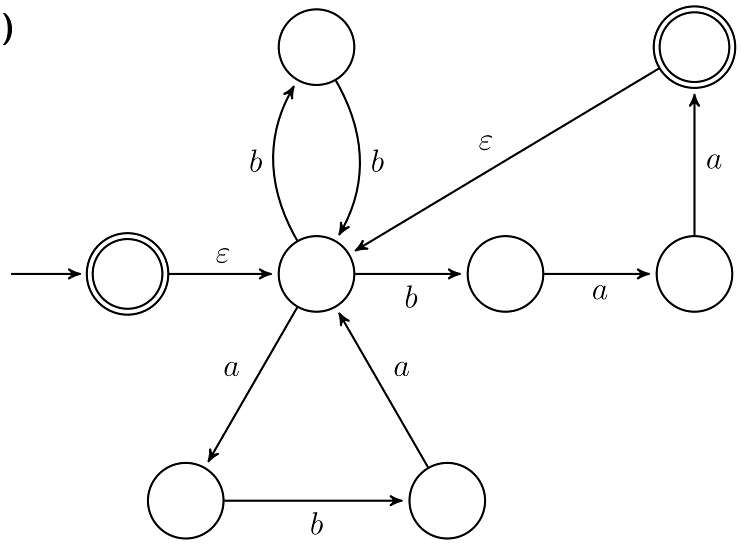
4–20)



4–21)



4–22)



Week 5

Pumping Lemma

5.1 Intuitive Idea Behind the Lemma

We have seen different ways to characterize a regular language. We can find a DFA, an NFA, or a regular expression. If it is impossible to find these, then the language is not regular.

For example, the languages over $\Sigma = \{0, 1\}$:

$$\begin{aligned} A &= \left\{0^n 1^n \;\middle|\; n \geqslant 0\right\} \\ B &= \left\{w \;\middle|\; w \text{ has an equal number of } 0\text{'s and } 1\text{'s}\right\} \end{aligned}$$

are non-regular, because we can find neither an automaton nor a regular expression for them.

The basic reason is that the machine needs infinitely many states to keep the number of symbols in memory. But it has finitely many states by definition. Also, the regular operation $(\ )^*$ cannot keep track of numbers.

Clearly, we need a better and more systematic way to determine whether a language is regular or not. That's what the pumping lemma does.

The basic idea behind the pumping lemma is this:

If an NFA accepts a long string, it must loop somewhere during the computation.

Here, long string means w where $|w| \geqslant n$ and n is the number of states of the NFA.

Let $\Sigma = \{a, b, c\}$ and consider an NFA with 5 states. It can accept strings of length at most 4 without looping. Suppose it accepts the string $abca$. That may look like this:

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[r, "a"] & q_1 \arrow[r, "b"] & q_2 \arrow[r, "c"] & q_3 \arrow[r, "a"] & q_4
\end{tikzcd}$$

Here, we are just showing the path taken by this string. The NFA has 5 states but it may have many more transitions.

Now consider an accepted string of length 5, $abaca$. Its computation may look like:

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[r, "a"] & q_1 \arrow[r, "b"] & q_2 \arrow[r, "c"] \arrow[loop above, "a"] & q_3 \arrow[r, "a"] & q_4
\end{tikzcd}$$

For an accepted string of length 6, $abccca$ we may have:

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[r, "a"] & q_1 \arrow[r, "b"] & q_2 \arrow[r, bend right, "c"'] & q_3 \arrow[l, bend right, "c"'] \arrow[r, "a"] & q_4
\end{tikzcd}$$

The computation of these strings of length $5, 6$ or more *must* contain some duplicate states. They cannot have a linear path that does not cross itself the way a string w with $|w| \leqslant 4$ does.

Now, although we know very little about the structure of the machine and the language it recognizes, still, we can confidently claim something:

If it turns around a loop once, it can turn around the same loop twice, three times, or any other number of times. It can also do zero turns.

That is, it may not enter the loop at all. If the rest of the string is identical, the result of the computation would still be accept.

For example, based on the acceptance of the string of length 5, we can claim that any string of the form

$$aba^nca$$

must be accepted. Similarly, based on the acceptance of the string of length 6, we can claim that any string of the form

$$abc(cc)^na$$

must be accepted.

Note that $n \geqslant 0$.

This operation of copying a part of a string and pasting it n times is called **pumping**. In a nutshell, pumping lemma says any sufficiently long string of a regular language can be pumped.

5.2 The Pumping Lemma

For any regular language L, there exists an integer p such that if

$$s \in L \quad \text{and} \quad |s| \geqslant p$$

then s may be divided into three pieces

$$s = xyz$$

satisfying the following conditions:

- $|xy| \leqslant p$,
- $|y| \geqslant 1$,
- for each $k \geqslant 0, \quad xy^k z \in L$.

Here p is called the pumping length. Note that y^k means that k copies of y are concatenated together and $y^0 = \varepsilon$.

Sketch of Proof: Let p be the number of states of the NFA that recognizes L. Let

$$s = s_1 s_2 \cdots s_n$$

be a string with $n \geqslant p$. At least two distinct inputs must result in the same state by the pigeonhole principle. Let's call the index of these inputs i and j. In other words, the automaton is in the same state after inputs s_i and s_j. Furthermore, $i \neq j$.

$$x = s_1 s_2 \cdots s_i, \qquad y = s_{i+1} s_{i+2} \cdots s_j, \qquad z = s_{j+1} s_{j+2} \cdots s_n$$

The string y takes the NFA from the same state to the same state and the automaton accepts xyz therefore it must accept $xy^k z$.

Also, $i \neq j$ therefore $|y| \geqslant 1$ and $|xy| \leqslant p$.

For example, suppose $p = 6,\ |s| = 9$. It is obvious that there are at least two symbols in s which result in the same state during computation. Suppose these are s_4 and s_7 and they result in q_3.

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[r, dashed, "x = s_1 s_2 s_3 s_4"] & q_3 \arrow[loop above, dashed, "y = s_5 s_6 s_7"] \arrow[r, dashed, "z = s_8 s_9"] & \boxed{q_5}
\end{tikzcd}$$

To summarize in another way: If L is regular, $s \in L$ and length of s is p or greater, then, s can be separated into three pieces:

$$\boxed{\qquad\qquad s \qquad\qquad}$$
$$\boxed{\quad x \quad}\boxed{\quad y \quad}\boxed{\quad z \quad}$$

such that:

1. Length of $\boxed{\quad x \quad}\boxed{\quad y \quad}$ is at most p.

2. Length of $\boxed{\quad y \quad}$ is at least 1.

3. Furthermore

$$\boxed{\quad x \quad}\boxed{\quad z \quad} \in L$$
$$\boxed{\quad x \quad}\boxed{\quad y \quad}\boxed{\quad z \quad} \in L$$
$$\boxed{\quad x \quad}\boxed{\quad y \quad}\boxed{\quad y \quad}\boxed{\quad z \quad} \in L$$
$$\boxed{\quad x \quad}\boxed{\quad y \quad}\boxed{\quad y \quad}\boxed{\quad y \quad}\boxed{\quad z \quad} \in L$$
$$\vdots$$

5.3 Applications

Example 5–1: Show that $L = \{0^n1^n \mid n \geqslant 0\}$ is non-regular.

Solution: Be careful about the logical structure of the pumping lemma. The following are equivalent:

1. L is regular $\Rightarrow$ All strings in L longer than p can be pumped.

2. There exists a string in L longer than p that cannot be pumped $\Rightarrow$ L is not regular.

All we have to do is find a single string that cannot be pumped. Let's choose this test string as:

$$s = 0^p1^p$$

where p is the pumping length. This makes sure that $|s| \geqslant p$. Now, let's separate it into three pieces:

$$0^p1^p = xyz$$

What is y? There are two possibilities only:

- y consists of a single type of symbol. ($y = 1^k$ or $y = 0^k$). Then, $xy^2z \notin L$ because the number of 0's and 1's in this string are not equal.

- y contains both 0's and 1's. Once again $xy^2z \notin L$ because symbols will be out of order. The resulting string will not be of the form $0^n\,1^n$.

The string $s \in L$ cannot be pumped so L is not regular.

Example 5–2: Show that the following language is not regular:

$$L = \{0^n 1^{2n} 0^n \mid n \geqslant 0\}$$

Solution: Assume L is regular. Then, there is a pumping length p. Let's choose the test string as $s = 0^p\,1^{2p}\,0^p$. The length of s is greater than p so we should be able to pump it.

$$s = 0^p\,1^{2p}\,0^p = xyz$$

Here, the question is: What does y contain? Does it contain a single type of symbol or both types?

- If y consists of 1's only, for example, i of them:

  $$xy^2z = 0^p\,1^{2p+i}\,0^p$$

  $$xy^kz = 0^p\,1^{2p+(k-1)i}\,0^p$$

- If y consists of 0's only, for example, i of them:

  $$xy^2z = 0^{p+i}\,1^{2p}\,0^p$$

  $$xy^kz = 0^{p+(k-1)i}\,1^{2p}\,0^p$$

  Clearly, none of these strings belong to the language L.

- If y contains both 0's and 1's, then the resulting string is not of the form $0^n\,1^{2n}\,0^n$ (because symbols are out of order) so again $xy^iz \notin L$.

Therefore s cannot be pumped. So L is not regular.

Example 5–3: Show that the following language is not regular:

$$L = \left\{ a^{n-2} b^{n+2} \mid n \geqslant 2 \right\}$$

Solution: Assume L is regular and let pumping length be p.

Consider the test string $s = a^p b^{p+4}$.

Let's try to find strings x, y, z such that $s = xyz$ and the requirements of pumping lemma are satisfied.

1. y consists of a's only: In this case, $xy^kz \notin L$.

   The relationship between number of a's and b's is broken. For example,
   $y = a^2 \quad \Rightarrow \quad xyyz = a^{p+2}b^{p+4} \notin L.$

2. y consists of b's only: In this case, once again $xy^kz \notin L$ for the same reason. For example,

   $y = b \quad \Rightarrow \quad xyyz = a^p b^{p+5} \notin L.$

   Also, $|xy| > p$ which is against pumping lemma, condition 1.

3. y consists of both a's and b's: In this case, the relationship between number of a's and b's may hold. But once again, $xy^kz \notin L$ because the order of the symbols is wrong. We have a's after b's. For example,

   $y = ab \quad \Rightarrow \quad xyyz = a^p bab^{p+4} \notin L.$

   Or, $y = aaabb \quad \Rightarrow \quad xyyz = a^p bbaaab^{p+4} \notin L.$

It is not possible to pump $s = a^p b^{p+4}$ in any way.

Therefore L is NOT regular by pumping lemma.

Example 5–4: Consider the following languages over the alphabet $\Sigma = \{a, b, c\}$:

$$\begin{aligned} A &= \left\{a^n\, b^m\, c^{nm} \;\middle|\; n, m \geqslant 0\right\} \\ B &= \left\{a^{2n}\, b^m\, c^{k+1}\, a^2 \;\middle|\; n, m, k \geqslant 0\right\} \end{aligned}$$

One of these languages is regular. Find a regular expression for it. The other is not regular. Prove that it is not regular using pumping lemma.

Solution: B is regular. We can describe it by the regular expression

$$(aa)^*\, b^*\, c^+\, aa$$

Suppose A is regular, assume the pumping length to be p. Consider the string

$$s = a^p\, b^p c^{p^2}$$

Clearly, $|s| > p$. By pumping lemma, we should be able to pump it.

$$s = xyz$$

We also know that $|xy| \leqslant p$. Therefore y consists of a's only. If $y = a^i$, then

$$xyyz = a^{p+i}\, b^p c^{p^2} \notin A$$

Therefore our assumption is wrong. s cannot be pumped, so A is not regular.

Example 5–5: Show that $L = \left\{ 1^{n^2} \mid n \geqslant 0 \right\}$ is non-regular.

Solution: Assume L is regular. Let p be the pumping length. Consider the string

$$s = 1^{p^2} \in L$$

Note that $|s| \geqslant p$. Let's separate this string into three parts:

$$s = xyz$$

It is clear that $y = 1^k$ for some $k \leqslant p$. Pump it once. Obtain

$$xy^2z = 1^{p^2+k}$$

Is this string a part of the language? Is $p^2 + k$ a perfect square? The next square is

$$(p+1)^2 = p^2 + 2p + 1$$

The difference between the two is $2p + 1$ but we know that

$$k < 2p + 1$$

In other words $p^2 + k$ cannot be a perfect square.

$$\Rightarrow \quad xy^2z \notin L$$

The string s cannot be pumped therefore L is not regular.

Example 5–6: Show that $L = \left\{1^{n!} \;\middle|\; n \geqslant 0\right\}$ is nonregular.

Solution: Assume L is regular. Let p be the pumping length. Consider the string

$$s = 1^{p!} \in L$$

Note that $|s| \geqslant p$. Let

$$s = xyz$$

It is clear that $y = 1^k$ for some $k \leqslant p$. Pump it once. Obtain

$$xy^2z = 1^{p!+k}$$

What is $p! + k$? Is it the factorial of another number?

The next factorial is $(p+1)!$. The difference between the two is:

$$(p+1)! - p! = pp!$$

Clearly:

$$k \leqslant p < pp!$$

In other words

$$p! + k < (p+1)!$$

so it cannot be a factorial.

$$\Rightarrow \quad xy^2z \notin L$$

The string s cannot be pumped therefore L is not regular.

Example 5–7: Show that the language

$$L = \left\{ w \mid w = w^R \right\}$$

over the alphabet $\Sigma = \{0, 1\}$ is non-regular.

Here, w^R shows the reverse of the string w. Therefore, L is the language of palindromes. For example,

$$\begin{array}{llll} x = 11101, & x^R = 10111 & \Rightarrow & x \notin L. \\ y = 1100011, & y^R = 1100011 & \Rightarrow & y \in L. \end{array}$$

Solution: Suppose L is regular. Let the pumping length be p. Choose the test string as:

$$s = 0^p 1 0^p$$

We know that

$$s = xyz \quad \text{and} \quad |xy| \leqslant p$$

therefore y consists of 0's only.

$$y = 0^k \quad \text{where} \quad k \leqslant p$$

$$s = xyz \in L$$

but

$$xyyz = 0^{p+k} 1 0^p \notin L$$

for any possible choice of y. So there is a contradiction. L is non-regular by pumping lemma.

Example 5–8: Show that the following language over $\Sigma = \{a, b\}$ is non-regular:

$$L = \left\{a^n\, b^m \;\middle|\; n > m \geqslant 0\right\}$$

Solution: Choose the test string as

$$s = a^{p+1}\, b^p$$

If $xyz = s$ and $|xy| \leqslant p$, then y consists of a's only. In other words $y = a^i$.

$$\Rightarrow \quad xyyz = a^{p+i+1}\, b^p \in L$$

because there are more a's than b's. Similarly for xy^3z, xy^4z and others.

We have to be careful here because up to now, we could not find any contradiction with pumping lemma.

Here, we have to pump down. Don't forget that pumping lemma is also correct for $k = 0$, not only for positive powers.

As usual, y must be a^i, but now, consider

$$xy^0z = xz = a^{p-i+1}\, b^p \notin L$$

This string is not part of the language because the number of a's is fewer than or equal to the number of b's.

So L is not regular by pumping lemma.

Example 5–9: Show that the following language over $\Sigma = \{0, 1\}$ is non-regular:

$$A = \left\{0^n\, 1^m \;\middle|\; n, m \geqslant 0, \quad n \neq m.\right\}$$

Solution:

- It is difficult to choose the correct test string for the pumping lemma in this example. For example, if $p = 3$, the test strings $0^2 1^3,\ 0^4 1^3,\ 0^6 1^5$, etc. can be pumped by choosing $x = \varepsilon,\ y = 0^2$.

  Similarly, $0^4 1^8,\ 0^4 1^2,\ 0^5 1^9$, etc. can be pumped by choosing $x = \varepsilon,\ y = 0^3$.

  But if we start with $0^3 1^9$ then we can complete the proof. In other words, an appropriate test string is:

  $$s = 0^p\, 1^{p+p!}$$

  Details are left to the reader.

- A much more elegant proof is this:

  Consider the language $\left\{0^* 1^*\right\}$, in other words

  $B = \left\{0^n\, 1^m \;\middle|\; n, m \geqslant 0\right\}$. This is clearly regular.

  Assume the given language A is also regular. Then its complement $\overline{A}$ and also

  $$C = \overline{A} \cap B$$

  must be regular. But we know that

  $$C = \left\{0^n 1^n \;\middle|\; n \geqslant 0\right\}$$

  and it is non-regular. We have a contradiction.

  Therefore A must be non-regular.

EXERCISES

Consider the following languages over the alphabet $\Sigma = \{a, b, c\}$. If they are regular, find a regular expression representing them. Otherwise, prove that they are not regular using pumping lemma.

5–1) $L = \{a^{2n}\, b^m\, c \mid n, m \geqslant 0\}$

5–2) $L = \{a^{2n}\, b\, c^n \mid n \geqslant 0\}$

5–3) $L = \{a^n\, b^m\, c^n \mid n, m \geqslant 0\}$

5–4) $L = \{a^{3n+2}\, b^{m+1}\, c^k \mid n, m \geqslant 0, \quad k = 3 \quad \text{or} \quad k = 5.\}$

5–5) $L = \{ww \mid w \in \Sigma^*\}$

5–6) $L = \{a^n\, b^m\, c^k \mid m + n + k = 100\}$

5–7) $L = \{a^n\, b^m \mid 0 \leqslant n < m\}$

5–8) $L = \{a^n\, b^m\, c^{n+m} \mid n, m \geqslant 0\}$

Consider the following languages. If they are regular, find a regular expression representing them. Otherwise, prove that they are not regular.

5–9) $L = \{1^n \mid n \text{ is prime.}\}$

5–10) $L = \{w \mid w \text{ contains equal number of } 0\text{'s and } 1\text{'s.}\}$

5–11) $L = \{0^n 1^m \mid 2n < m\}$

5–12) $L = \{0^n 1^m \mid 2n > m\}$

5–13) $L = \{a^n b^m \mid \text{Both } n \text{ and } m \text{ are odd.}\}$

5–14) $L = \{a^k b^m c^n \mid k < m < n\}$

5–15) $L = \{a^k b^m c^n \mid k \geqslant 2, \quad m \geqslant 0, \quad n \leqslant 2.\}$

What happens if we try pumping lemma on a regular language? Try to find x, y, z satisfying pumping lemma's conditions for strings from the following languages:

5–16) $A = \{w \mid w \text{ starts with } 111.\}$

5–17) $B = \{w \mid w \text{ ends with } 111.\}$

5–18) $C = \{w \mid w \text{ contains } 111.\}$

5–19) $D = \{w \mid w \text{ has odd length.}\}$

5–20) $E = \{w \mid w \text{ starts and ends with the same symbol.}\}$

ANSWERS

5–1) Regular: $(aa)^* \, b^* \, c$

5–2) Non-Regular. Choose the test string as $s = a^{2p} \, b \, c^p$.

5–3) Non-Regular. Choose the test string as $s = a^p \, b \, c^p$.

$$xyz = s \quad \text{and} \quad |xy| \leqslant p \quad \Rightarrow \quad y = a^k$$

$$\Rightarrow \quad xyyz = a^{p+k} \, b \, c^p \notin L$$

because there are more a's than c's.

5–4) Regular: $aa(aaa)^* \, bb^* \, (ccc \cup ccccc)$

5–5) Non-Regular. Choose the test string as $s = a^p \, b \, a^p \, b$.

5–6) Regular because the language has finitely many strings. The regular expression is too long.

5–7) Non-Regular. Choose the test string as $s = a^p \, b^{p+1}$.

5–8) Non-Regular. Choose the test string as $s = a^p \, b^p \, c^{2p}$.

5–9) Non-Regular. Choose the test string as $s = 1^k$ where k is prime and $k > p$. Suppose we decompose s as

$$xyz = s \quad \text{and} \quad y = 1^i$$

Now consider

$$\Rightarrow \quad xy^{k+1}z = 1^k\,1^{ik} = 1^{k(i+1)}$$

$k(i+1)$ is not prime $\quad \Rightarrow \quad xy^{k+1}z \notin L$

5–10) Non-Regular. Choose the test string as $s = 0^p 1^p$.

5–11) Non-Regular. Choose the test string as $s = 0^p 1^{2p+1}$. Pump up.

5–12) Non-Regular. Choose the test string as $s = 0^p 1^{2p-1}$. Pump down.

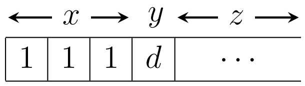
5–13) Regular: $a\big(aa\big)^* b\big(bb\big)^*$

5–14) Non-Regular. Choose the test string as $s = a^p b^{p+1} c^{p+2}$.

5–15) Regular: $aaa^*\, b^*\, (\varepsilon \cup c \cup cc)$

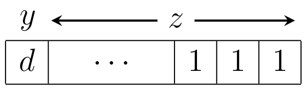
In the following d denotes a single digit 0 or 1 and $\cdots$ denotes a substring that may or may not be ε.

5–16) $p = 4$, $s = 111d\cdots$



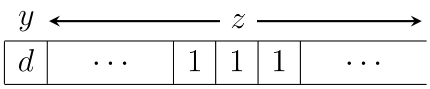
Note that $|xy| \leqslant 4$ and $xy^kz \in A$.

5–17) $p = 4$, $s = d\cdots 111$

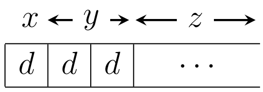


Note that $x = \varepsilon$, $|xy| = 1 \leqslant 4$ and $xy^kz \in B$.

5–18) Once again, $p = 4$. If s starts with 111, use the partition given in *5–16)*. Otherwise, use:

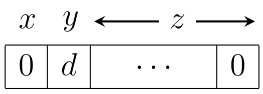


5–19) $p = 3$, $s = ddd\cdots$

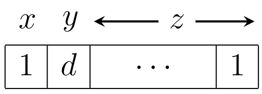


Note that $|xy| \leqslant 3$ and $xy^kz \in D$.

5–20) $p = 3$, $s = 1d\cdots 1$ or $s = 0d\cdots 0$.



or



Week 6

Context-Free Grammars

6.1 Grammars and Languages

Let's start with a sample grammar:

$$\begin{array}{rcl} S & \to & ABCD \\ A & \to & \text{I} \mid \text{You} \\ B & \to & \text{will} \mid \text{can} \mid \text{must} \\ C & \to & \text{not} \mid \varepsilon \\ D & \to & \text{continue} \mid \text{stop} \end{array}$$

Here, S, A, B, C, D are variables. *You*, *not*, *stop*, etc. are terminals. We start with variable S and interpret the symbol $\mid$ as *or*.

The idea is to use the given substitutions until we obtain a string made of terminals only.

Using the given grammar, we can obtain 24 sentences (strings). For example,

- I will not continue.
- You can not stop.
- You must continue.

Formal Definition of a CFG:
A **Context-Free Grammar (CFG)** is 4-tuple (V, Σ, R, S) where

- V is a finite set called the **variables**,
- Σ is a finite set, disjoint from V, called the **terminals**,
- R is a finite set of **rules**, also called **productions**, with each rule connecting a variable and a string of variables and terminals.
- $S \in V$ is the **start variable**.

For example,

$$\begin{array}{rcl} S & \rightarrow & 0A1 \\ A & \rightarrow & 0A \mid 1A \mid \varepsilon \end{array}$$

is a context-free grammar where:

- Variables are: S and A.
- Terminals are: 0 and 1.

A typical derivation of a string is:

$$\begin{array}{rcl} S & \rightarrow & 0A1 \\ & \rightarrow & 00A1 \\ & \rightarrow & 001A1 \\ & \rightarrow & 001\varepsilon 1 \\ & \rightarrow & 0011 \end{array}$$

A language is called a **Context-Free Language (CFL)** if there is a context-free grammar G with $L = L(G)$.

The language in the example above can be described as:
Strings of length at least two, starting with 0 *and ending with* 1.

Example 6–1: Describe the language generated by the following CFG over $\Sigma = \{0,1\}$:

$$S \rightarrow 0S1 \bigm| \varepsilon$$

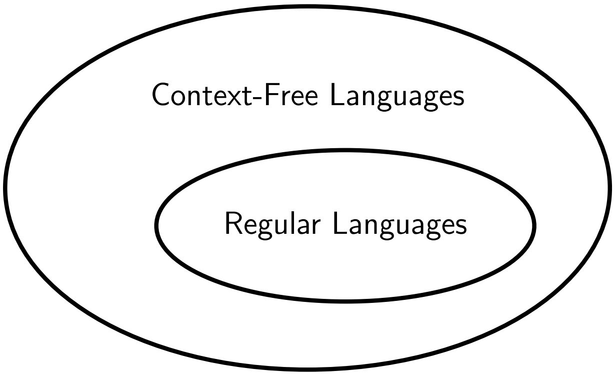
Solution: If we use the first rule of substitution repeatedly, we will obtain:

$$\begin{array}{rcl} S & \rightarrow & 0S1 \\ & \rightarrow & 00S11 \\ & \vdots & \\ & \rightarrow & 0^iS1^i \\ & \vdots & \end{array}$$

We have to use the rule $S \rightarrow \varepsilon$ at some point to end the iteration. At that point, we obtain the string 0^n1^n. Therefore the language generated by this CFG is:

$$L = \bigl\{w \bigm| w = 0^n1^n, \quad n \geqslant 0\bigr\}$$

We have seen previously that this language was not regular. Now we see that it is context-free. We will later prove that every regular language is context-free.



Example 6–2: The following are some typical regular languages over the alphabet $\Sigma = \{0, 1\}$. Find CFGs that generate them:

1. $\{w \mid w$ starts with $00.\}$
2. $\{w \mid w$ ends with $11.\}$
3. $\{w \mid w$ contains $101.\}$
4. $\{w \mid |w| = 4.\}$
5. $\{w \mid |w| \leqslant 4.\}$
6. $\{w \mid |w| \equiv 0 \pmod 3.\}$
7. $\{w \mid w$ starts and ends with the same symbol.$\}$

Solution:

1. $$\begin{aligned} S &\rightarrow 00A \\ A &\rightarrow 0A \mid 1A \mid \varepsilon \end{aligned}$$

2. $$S \rightarrow 0S \mid 1S \mid 11$$

3. $$\begin{aligned} S &\rightarrow A101A \\ A &\rightarrow 1A \mid 0A \mid \varepsilon \end{aligned}$$

4. $$\begin{aligned} S &\rightarrow AAAA \\ A &\rightarrow 0 \mid 1 \end{aligned}$$

5. $$\begin{aligned} S &\rightarrow AAAA \\ A &\rightarrow 0 \mid 1 \mid \varepsilon \end{aligned}$$

6. $$\begin{aligned} S &\rightarrow AAAS \mid \varepsilon \\ A &\rightarrow 0 \mid 1 \end{aligned}$$

7. $$\begin{aligned} S &\rightarrow 0A0 \mid 1A1 \mid 0 \mid 1 \\ A &\rightarrow 0A \mid 1A \mid \varepsilon \end{aligned}$$

A **palindrome** is a string that is equal to its reverse. In other words, it must be the same string whether we read it from left to right or right to left. For example,

$$abba,\ aa,\ abaaaba,\ bbbabbb$$

are some palindromes. The reverse of a string is denoted by w^R so we can characterize the language of palindromes as follows:

$$L = \{w \mid w = w^R\}$$

Previously, we have proved that this language is not regular using pumping lemma. We can also easily see that designing an NFA recognizing that language is impossible because it would need an infinite amount of memory, in other words, infinitely many states.

Is this language context-free?

Example 6–3: Design a CFG that generates palindromes of even length over $\Sigma = \{a, b\}$.

Solution: $S \to aSa \mid bSb \mid \varepsilon$

Example 6–4: Design a CFG that generates palindromes of odd length over $\Sigma = \{a, b\}$.

Solution: $S \to aSa \mid bSb \mid a \mid b$

Example 6–5: Design a CFG that generates palindromes over $\Sigma = \{a, b\}$.

Solution: $S \to aSa \mid bSb \mid a \mid b \mid \varepsilon$

Example 6–6: Show that the CFG

$$S \to aSa \mid bSb \mid a \mid b \mid \varepsilon$$

generates the CFL of palindromes over $\Sigma = \{a, b\}$.

Solution: For simple grammars, we can easily see what kind of strings are generated at a glance, but for more complicated cases, we need a proof. We will use induction in such proofs.

Let's remember mathematical induction with an example. Suppose we want to prove that

$$1 + 3 + 5 + \cdots + (2n - 1) = n^2$$

for all positive integers n. We check the statement for $n = 1$.

$$1 = 1^2 = 1$$

This is obviously correct. Now we assume it is correct for $n = 1, 2, \ldots k$ and check if it is also correct for $n = k + 1$:

$$1 + 3 + 5 + \cdots + (2k - 1) = k^2$$

Add the next term $(2k + 1)$ to both sides:

$$\begin{aligned} 1 + 3 + \cdots + (2k - 1) + (2k + 1) &= k^2 + 2k + 1 \\ &= (k + 1)^2 \end{aligned}$$

This completes the proof.

Let's call the language generated by the above grammar L. The proof has two parts.

Part I: If w is a palindrome, then $w \in L$.
Suppose w is a palindrome. Induction is on the length of w.

1. If $|w| = 0$ or $|w| = 1$, w must be ε or a or b, therefore $w \in L$ because there are productions:

$$S \to \varepsilon, \quad S \to a, \quad S \to b$$

2. Now assume all palindromes with $|w| \leqslant k$ are in L. Let w be a palindrome with $|w| = k + 1$. It is clear that there exists a string v where $w = 1v1$ or $w = 0v0$ and $|v| = k - 1$. This means v is a palindrome and therefore $v \in L$. So $w \in L$.

Part II: If $w \in L$, then w is a palindrome.
Suppose we obtain w using grammar productions in n steps. Induction is on n.

1. If $n = 1$, w must be ε or a or b, therefore w is a palindrome.

2. Now assume all strings obtained after k steps are palindromes. Let $n = k + 1$. The first step must be

$$S \to 0S0 \quad \text{or} \quad S \to 1S1$$

   After k more steps we obtain $w = 0v0$ or $w = 1v1$ where v is a string obtained in k steps. Therefore v is a palindrome and w is also a palindrome.

Example 6–7: What language does the following CFG generate?

$$S \to aSb \mid SS \mid \varepsilon$$

Solution: This is a language of strings of a's and b's such that:

- The number of a's and b's in the string are the same.
- At any intermediate point in the string, the number of a's is greater than or equal to the number of b's.

(We can think of those as open and close parentheses.)

Example 6–8: Give a CFG generating L over $\Sigma = \{0, 1\}$:

$$L = \{w \mid w \text{ is of odd length and contains at least two 0's.}\}$$

Solution: We know that there are two 0's. There must be three other pieces of substring: $S \to ?\,0\,?\,0\,?$

These substrings denoted by "?" may or may not be ε. What is important is the length. Either one of them is odd, two of them are even or all three are odd.

Let's use the symbol A for odd and B for even length strings. Then the grammar is:

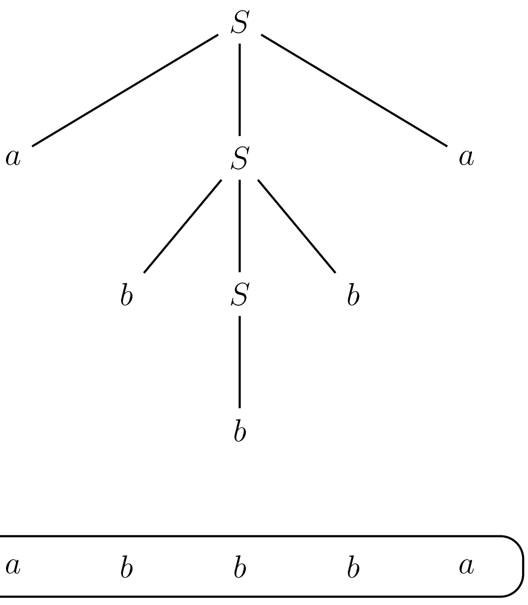
$$\begin{aligned} S &\to A0A0A \mid A0B0B \mid B0A0B \mid B0B0A \\ A &\to 0B \mid 1B \\ B &\to \varepsilon \mid 00B \mid 01B \mid 10B \mid 11B \end{aligned}$$

6.2 Parse Trees

For a CFG (V, Σ, R, S) a **parse tree** (or derivation tree) is a tree with the following properties:

- The root is labeled S.
- Each interior vertex is labeled by a variable.
- Each leaf is labeled by either a terminal or ε.
- A leaf labeled by ε must be the only child of its parent.
- If a vertex with label A has children with labels $B_1, B_2, \ldots, B_n$ then the grammar must have a production $A \to B_1 B_2 \ldots B_n$.

For example, the derivation of the string $abbba$ in the palindrome grammar $S \to aSa \mid bSb \mid a \mid b \mid \varepsilon$ can be represented by:



Example 6–9: Consider the following grammar:

$$\begin{array}{rcl} S & \rightarrow & SS \mid A \\ A & \rightarrow & aAa \mid B \\ B & \rightarrow & bB \mid \varepsilon \end{array}$$

Let's call the language generated by this grammar L. Now, consider the string:

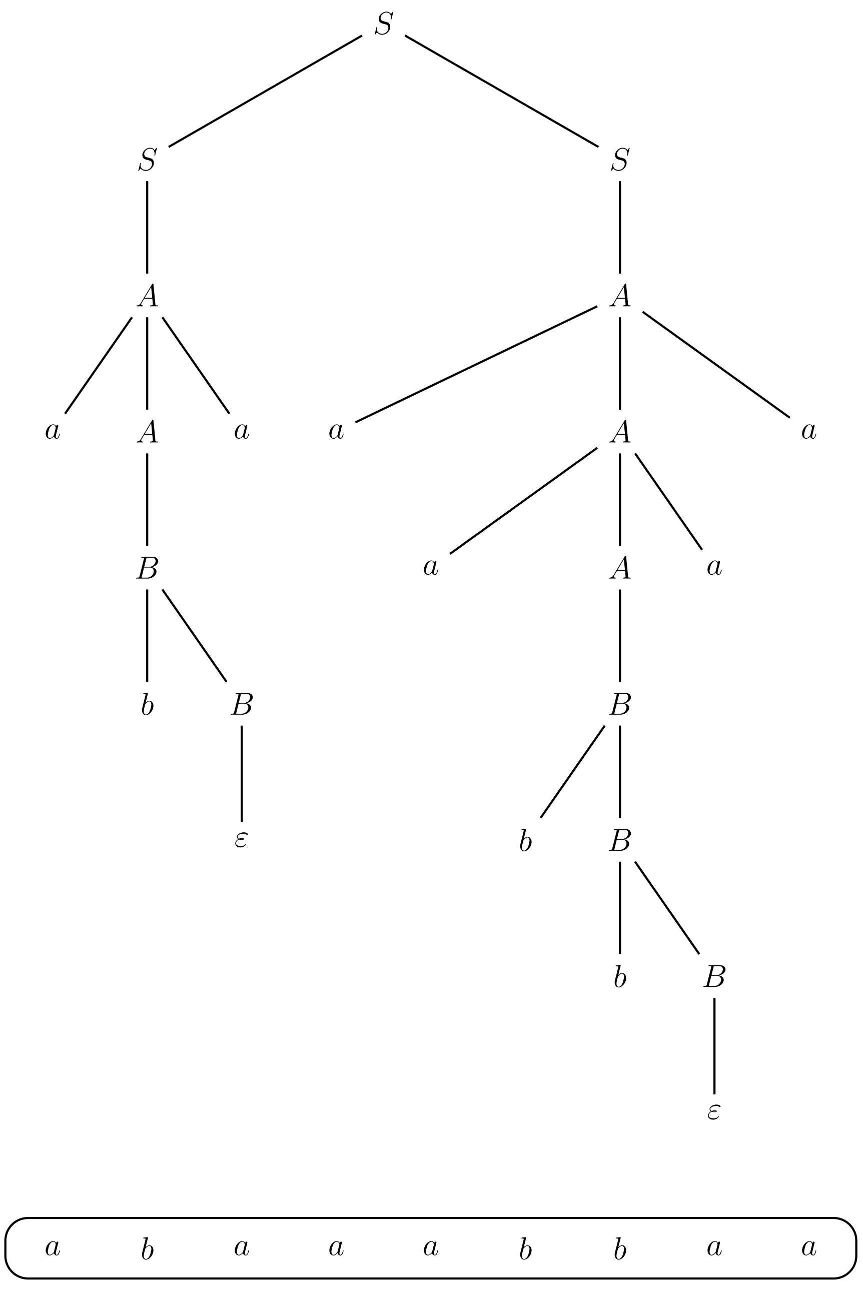
$$w = abaaabbaa$$

Show that $w \in L$.

Solution: To show that this string belongs to L, we must give a derivation:

$$\begin{array}{rcl} S & \rightarrow & SS \\ & \rightarrow & AA \\ & \rightarrow & aAaaAa \\ & \rightarrow & aBaaaAaa \\ & \rightarrow & abBaaaBaa \\ & \rightarrow & abaaabBaa \\ & \rightarrow & abaaabbBaa \\ & \rightarrow & abaaabbaa \end{array}$$

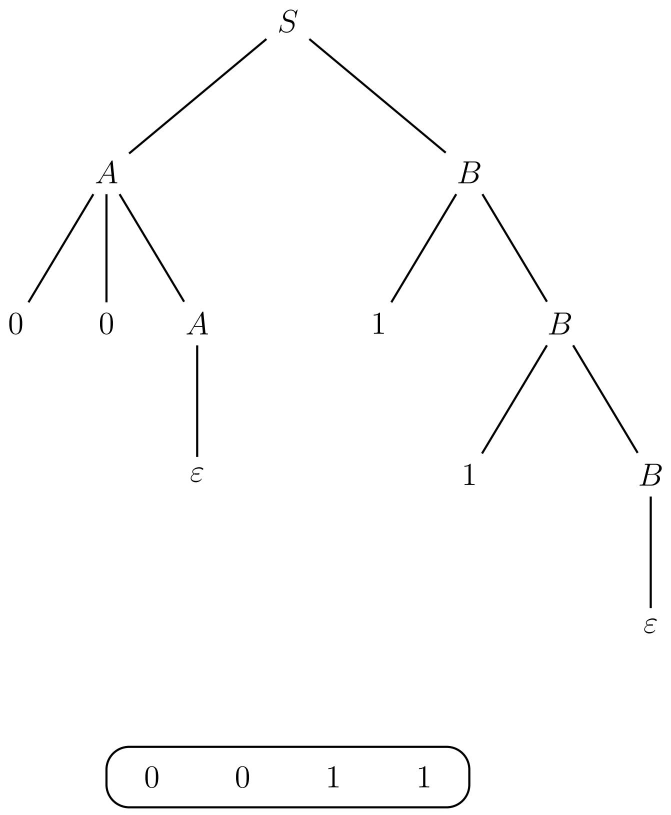
It is possible to show this derivation in a more visual way using a parse tree.


S
S
S
A
A
a
A
a
a
A
a
B
a
A
a
b
B
B
ε
b
B
b
B
ε
a b a a a b b a a

Example 6–10: Find the parse tree for the string 0011 using the grammar:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow 00A \mid \varepsilon \\ B &\rightarrow 1B \mid \varepsilon \end{aligned}$$

Solution:



6.3 Ambiguity

It is possible to derive a string in a grammar in different ways. For example, using the grammar in the previous question, the string 0011 can be derived as:

$$S \rightarrow AB \rightarrow 00AB \rightarrow 00B \rightarrow 001B \rightarrow 0011B \rightarrow 0011$$

or alternatively, as:

$$S \rightarrow AB \rightarrow A1B \rightarrow A11B \rightarrow A11 \rightarrow 00A11 \rightarrow 0011$$

A derivation of a string w in a grammar G is a **leftmost derivation** if at every step the leftmost remaining variable is the one replaced.

You can easily check that the first derivation above is a leftmost derivation and the second one is a rightmost derivation.

Also, check that they have exactly the same parse trees. In other words, there is not a fundamental difference between these two derivations.

Sometimes a grammar can generate the same string in several different ways. Such a string will have several different parse trees and thus several different meanings.

A CFG is **ambiguous** if there is a string which has:

- Two leftmost derivations, or
- Two different parse trees.

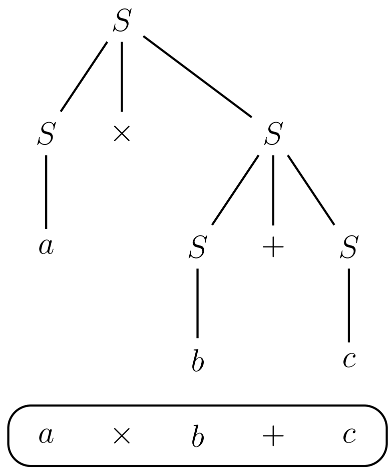
(Note that these two conditions are equivalent.)

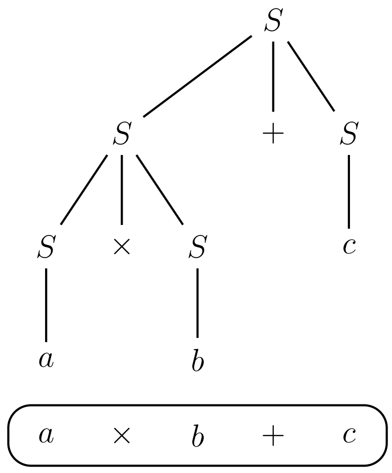
Ambiguity is an important problem in programming languages.

Example: The grammar

$$\begin{aligned} S &\to S \times S \\ S &\to S + S \\ S &\to (S) \mid a \mid b \mid c \end{aligned}$$

generates the string $a \times b + c$ in two different ways:





These derivations have different parse trees. Therefore this grammar is ambiguous. The string

$$a \times b + c$$

can be interpreted in two different ways.

On the other hand, the grammar:

$$\begin{array}{lcl} S & \to & S + A \mid A \\ A & \to & A \times B \mid B \\ B & \to & (S) \mid a \mid b \mid c \end{array}$$

generates the same language, but is not ambiguous.

Sometimes when we have an ambiguous grammar we can find an unambiguous grammar that generates the same language.

Some context-free languages, however, can be generated only by ambiguous grammars. Such languages are called **inherently ambiguous**.

For example, the language L defined over $\Sigma = \{a, b, c\}$ as:

$$L = \{w \mid w = a^k b^m c^n \quad \text{where} \quad k = m \quad \text{or} \quad k = n\}$$

is inherently ambiguous, because the string

$$a^n b^n c^n \in L$$

always has two distinct parse trees. This is more clear if you consider L as $L = L_1 \cup L_2$ where

$$L_1 = \{w \mid w = a^m b^m c^n\},$$

$$L_2 = \{w \mid w = a^n b^m c^n\},$$

Example 6–11: The following CFG is ambiguous:

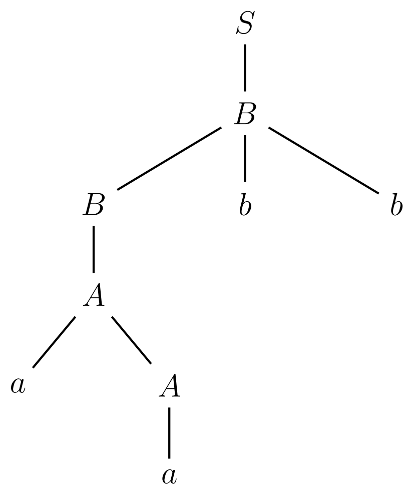
$$\begin{aligned} S &\to B \mid AB \\ A &\to aA \mid a \\ B &\to Bbb \mid A \end{aligned}$$

a) Consider the string $w = aabb$. Give two different leftmost derivations and corresponding parse trees.

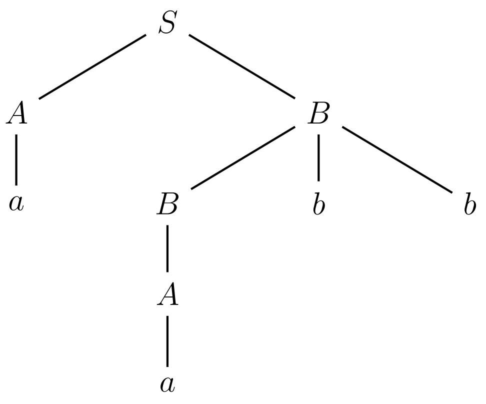
b) Modify the grammar such that it generates the same language but it is no longer ambiguous.

Solution: a)

$$\begin{aligned} S &\to B \\ &\to Bbb \\ &\to Abb \\ &\to aAbb \\ &\to aabb \end{aligned}$$



$$\begin{aligned} S &\to AB \\ &\to aB \\ &\to aBbb \\ &\to aAbb \\ &\to aabb \end{aligned}$$



b) We can easily see that the given grammar generates the language:

$$L = \{w \mid w = a^n b^{2m}, \quad n \geqslant 1,\ m \geqslant 0.\}$$

The following grammars are unambiguous and each of them generates the same language:

$$\begin{aligned} S &\to B \\ A &\to aA \mid a \\ B &\to Bbb \mid A \end{aligned}$$

$$\begin{aligned} S &\to AB \\ A &\to aA \mid a \\ B &\to Bbb \mid \varepsilon \end{aligned}$$

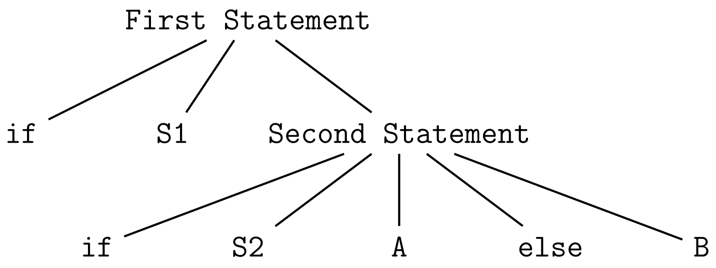
Example: Another example of ambiguity is dangling else. Consider the piece of code:

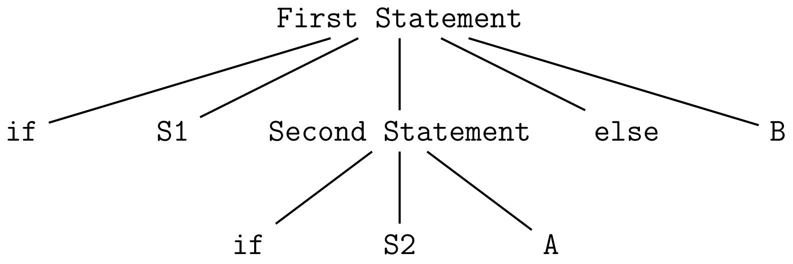
```
if S1 if S2 A else B
```

How should we understand this? There are two possibilities:

```
if S1                          if S1
   if S2                          if S2
      A                              A
   else                        else
      B                           B
```

Their parse trees are different:





EXERCISES

Find a CFG that generates the given language over the alphabet Σ:

6–1) $L = \{a^{2n}\, b^{2m+1}\, c^n \mid m, n \geqslant 0\}, \quad \Sigma = \{a, b, c\}.$

6–2) $L = \{a^n b^{n+2} \mid n \geqslant 0\}, \quad \Sigma = \{a, b\}.$

6–3) $L = \{a^n b^m \mid n < m\}, \quad \Sigma = \{a, b\}.$

6–4) $L = \{w \mid w$ is of odd length and contains more 0's than 1's.$\}$, $\Sigma = \{0, 1\}.$

6–5) $L = \{w \mid w$ is of even length and starts and ends with the same symbol.$\}, \quad \Sigma = \{0, 1\}.$

6–6) $L = \{w \mid |w| \geqslant 5\}, \quad \Sigma = \{0, 1\}.$

6–7) $L = \{w \mid |w| \leqslant 5\}, \quad \Sigma = \{0, 1\}.$

6–8) $L = \{a^k\, b^m\, c^n \mid k, m, n \geqslant 0,\ k + n = m\}, \quad \Sigma = \{a, b, c\}.$

6–9) $L = \{a^k\, b^m\, c^n \mid k, m, n \geqslant 0,\ k + n = 2m\}, \quad \Sigma = \{a, b, c\}.$

6–10) $L = \{a^k\, b^m\, c^n \mid m = k \quad \text{or} \quad m = n\}, \quad \Sigma = \{a, b, c\}.$

Describe the language generated by the following CFGs:

6–11)
$$\begin{aligned} S &\rightarrow aSc \mid T \mid \varepsilon \\ T &\rightarrow aTb \mid \varepsilon \end{aligned}$$

6–12)
$$\begin{aligned} S &\rightarrow aSbb \mid T \\ T &\rightarrow Tb \mid \varepsilon \end{aligned}$$

6–13)
$$\begin{aligned} S &\rightarrow aSbb \mid T \\ T &\rightarrow Tb \mid b \end{aligned}$$

6–14)
$$\begin{aligned} S &\rightarrow aST \mid \varepsilon \\ T &\rightarrow b \mid bb \mid bbb \end{aligned}$$

6–15)
$$\begin{aligned} S &\rightarrow BTA \\ T &\rightarrow abT \mid \varepsilon \\ A &\rightarrow a \mid \varepsilon \\ B &\rightarrow b \mid \varepsilon \end{aligned}$$

6–16)
$$\begin{aligned} S &\rightarrow TST \mid c \\ T &\rightarrow a \mid b \mid c \end{aligned}$$

6–17)
$$\begin{aligned} S &\rightarrow 0S1 \mid A \mid B \\ A &\rightarrow 0A \mid 0 \\ B &\rightarrow B1 \mid 1 \end{aligned}$$

6–18)
$$\begin{aligned} S &\rightarrow A \mid B \\ A &\rightarrow CCCA \mid \varepsilon \\ B &\rightarrow CCCCCB \mid \varepsilon \\ C &\rightarrow 0 \mid 1 \end{aligned}$$

Describe the language generated by the following CFGs:

6–19)
$$\begin{array}{lcl} S & \to & 0A0 \mid 1B1 \mid 0 \mid 1 \\ A & \to & TAT \mid 0 \\ B & \to & TBT \mid 1 \\ T & \to & 0 \mid 1 \end{array}$$

6–20)
$$\begin{array}{lcl} S & \to & 0S1 \mid 0S11 \mid 0S111 \mid T \\ T & \to & 000 \end{array}$$

The following grammars are ambiguous. Find a string that has two different leftmost derivations. If possible, find an equivalent unambiguous grammar.

6–21)
$$\begin{array}{lcl} S & \to & 0S \mid 1A \mid 0B0 \\ A & \to & 0A \mid \varepsilon \\ B & \to & 0B0 \mid 1 \end{array}$$

6–22)
$$\begin{array}{lcl} S & \to & AB \mid BA \\ A & \to & A0 \mid \varepsilon \\ B & \to & 1B \mid \varepsilon \end{array}$$

6–23)
$$\begin{array}{lcl} S & \to & AB \mid C \\ A & \to & 0A1 \mid \varepsilon \\ B & \to & 0B1 \mid \varepsilon \\ C & \to & 0C1 \mid D \\ D & \to & 1D0 \mid \varepsilon \end{array}$$

ANSWERS

6–1)
$$\begin{aligned} S &\to aaSc \mid T \\ T &\to bbT \mid b \end{aligned}$$

6–2)
$$\begin{aligned} S &\to aSb \mid A \\ A &\to bb \end{aligned}$$

6–3)
$$\begin{aligned} S &\to aSb \mid A \\ A &\to bA \mid b \end{aligned}$$

6–4)
$$\begin{aligned} S &\to T0T \\ T &\to 0T1 \mid 1T0 \mid 0T0 \mid TT \mid \varepsilon \end{aligned}$$

6–5)
$$\begin{aligned} S &\to 0A0 \mid 1A1 \mid \varepsilon \\ A &\to 00A \mid 01A \mid 10A \mid 11A \mid \varepsilon \end{aligned}$$

6–6)
$$\begin{aligned} S &\to AAAAAB \\ A &\to 0 \mid 1 \\ B &\to 0B \mid 1B \mid \varepsilon \end{aligned}$$

6–7)
$$\begin{aligned} S &\to AAAAA \\ A &\to 0 \mid 1 \mid \varepsilon \end{aligned}$$

6–8)
$$\begin{aligned} S &\to AC \\ A &\to aAb \mid \varepsilon \\ C &\to bCc \mid \varepsilon \end{aligned}$$

6–9) $$\begin{aligned} S &\rightarrow AC \mid aAbCc \\ A &\rightarrow aaAb \mid \varepsilon \\ C &\rightarrow bCcc \mid \varepsilon \end{aligned}$$

6–10) $$\begin{aligned} S &\rightarrow TC \mid AU \\ T &\rightarrow aTb \mid \varepsilon \\ U &\rightarrow bUc \mid \varepsilon \\ A &\rightarrow aA \mid \varepsilon \\ C &\rightarrow cC \mid \varepsilon \end{aligned}$$

6–11) $\{a^n\, b^m\, c^k \mid n = m + k\}$

6–12) $\{a^n\, b^m \mid 2n \leqslant m\}$

6–13) $\{a^n\, b^m \mid 2n < m\}$

6–14) $\{a^n\, b^m \mid n \leqslant m \leqslant 3n\}$

6–15) Strings that do not contain aa or bb. In other words, a's and b's must be alternating.

6–16) Strings of odd length where middle symbol is c.

6–17) $\{0^n\, 1^m \mid n \neq m\}$

6–18) $\{w \mid |w|$ is a multiple of 3 or $5.\}$

6–19) Strings of odd length where the first, middle and the last symbols are the same.

6–20) Mathematical description:

$$L = \left\{ 0^{n+3}\, 1^{m} \;\middle|\; n \geqslant 0, \;\; n \leqslant m \leqslant 3n \right\}$$

Verbal description:

- The strings of the language start with 0's and then continue with 1's. (There is no 0 after the first 1.)
- The number of 0's is at least three.
- The number of 1's is at least the number of 0's minus three.
- The number of 1's is at most the number of 0's times three minus nine.

6–21) $w = 010$ or $w = 00100$ have two different leftmost derivations. An equivalent unambiguous grammar is:

$$\begin{array}{lcl} S & \to & 0S \mid 1A \\ A & \to & 0A \mid \varepsilon \end{array}$$

6–22) $w = 00$ or $w = 111$ have two different leftmost derivations. An equivalent unambiguous grammar is:

$$\begin{array}{lcl} S & \to & \varepsilon \mid A \mid B \mid AB \mid BA \\ A & \to & A0 \mid 0 \\ B & \to & 1B \mid 1 \end{array}$$

6–23) $w = 0101$ or $w = 0^2 1^2 0^2 1^2$ have two different leftmost derivations. Grammar is inherently ambiguous.

Week 7

Different Forms of Grammars

7.1 Simplifying Grammars

Different grammars may produce the same language. For example, the following grammars are equivalent:

G_1	G_2
$S \to 00S0 \mid D \mid B$	$S \to 00S0 \mid A$
$A \to 1A \mid 1$	$A \to 1A \mid 1$
$B \to 0B \mid BB$	
$C \to 0D \mid B1$	
$D \to A$	

Both grammars generate the language

$$\left\{0^{2n}\,1^m\,0^n \;\middle|\; n \geqslant 0,\; m \geqslant 1\right\}$$

In other words $L(G_1) = L(G_2)$ but G_2 is much simpler. In G_1, the variables B, C, D are unnecessary. Can you see why?

Useless Symbols: There are grammars where some symbols (variables or terminals) will not be used in any possible derivation of a string. Therefore we can eliminate all productions containing them without changing the language.

We call such symbols **useless symbols**.

There are two types of useless symbols:

1. X is not **reachable**. That means, starting with S and using any combination of rules of the grammar, we will never see X on the right-hand side at any step.
2. Any intermediate formula that contains X will never give us a string, because:
   (a) X is **non-generating**. In other words, starting with X and using productions, we will never obtain a string of terminals.
   (b) All the intermediate formulas containing X also contain non-generating symbols.

For example, in the grammar

$$\begin{array}{lcl} S & \to & 0S0 \mid A \mid B \mid D \\ A & \to & 1A0 \mid 1 \mid ED \\ B & \to & 0B1 \mid \varepsilon \mid DED \\ C & \to & 01S \mid AB \\ D & \to & 0D0 \mid AD \\ E & \to & 101 \mid 010 \end{array}$$

A and B are reachable and generating, but:

- C is not reachable.
- D is non-generating.
- E is useless although it is reachable and generating. (That is Type 2b.) E is always together with D so it is effectively non-generating in some sense.

Productions containing a useless symbol are called **useless productions**. If we eliminate all the useless productions, we obtain a much simpler grammar:

$$\begin{array}{lcl} S & \rightarrow & 0S0 \mid A \mid B \\ A & \rightarrow & 1A0 \mid 1 \\ B & \rightarrow & 0B1 \mid \varepsilon \end{array}$$

To eliminate all useless symbols:

1. Eliminate all non-generating symbols. This will make Type 2b symbols unreachable.
2. Eliminate all unreachable symbols.

In the example, eliminate D because it is non-generating and then C and E because they are unreachable.

Note that eliminating first the unreachable and then non-generating symbols may not accomplish this, because the second step may create new unreachables.

In the example, if we eliminate C first (because it is unreachable) and then D, (because it is non-generating) E remains in the grammar although it is unreachable now. We need to check unreachables once more as a third step.

Example 7–1: Eliminate all useless symbols and productions from the following grammar:

$$\begin{array}{rcl} S & \rightarrow & 0S1 \mid SS \mid BD \mid ABC \mid \varepsilon \\ A & \rightarrow & 00 \mid 1 \mid S \mid CD \\ B & \rightarrow & 0B \mid DD \mid ADA \\ C & \rightarrow & 010 \mid 101 \\ D & \rightarrow & D11 \mid B \\ E & \rightarrow & 1S1 \mid ESE \end{array}$$

Solution: If we check all productions carefully, we see that:

- S, A and C are reachable and generating.
- E is unreachable.
- B and D are non-generating.

Therefore we have to eliminate variables B, D and productions involving them first.

$$\begin{array}{rcl} S & \rightarrow & 0S1 \mid SS \mid \varepsilon \\ A & \rightarrow & 00 \mid 1 \mid S \\ C & \rightarrow & 010 \mid 101 \\ E & \rightarrow & 1S1 \mid ESE \end{array}$$

In this new grammar, clearly, A, C and E are unreachable. If we eliminate them, we obtain the much simpler equivalent grammar:

$$S \rightarrow 0S1 \mid SS \mid \varepsilon$$

Unit Productions: A unit production is a rule of substitution of the form

$$A \to B$$

where both A and B are variables. ($A \to a$ is not a unit production.) Unit productions unnecessarily increase the number of steps in a derivation.

Suppose

$$B \to X_1 \mid X_2 \mid \cdots \mid X_n$$

represent the set of all productions starting with B. Here X_i denotes a string of variables and terminals. Then we can eliminate $A \to B$ and replace it by

$$A \to X_1 \mid X_2 \mid \cdots \mid X_n$$

without changing the grammar.

For example, by removing the unit production $S \to B$ from the grammar

$$\begin{array}{lcl} S & \to & 0A0 \mid B \\ A & \to & BB \mid 01 \\ B & \to & 0 \mid 1A \end{array}$$

we obtain the non-unit productions:

$$\begin{array}{lcl} S & \to & 0A0 \mid 0 \mid 1A \\ A & \to & BB \mid 01 \\ B & \to & 0 \mid 1A \end{array}$$

Considering there may be rules like $A \to B$ and $B \to C$, we may be introducing new unit rules (for example, $A \to C$) as we remove old ones. So it is better to find all variables that can be reached from a variable by unit productions and then remove all those productions at one step.

Example 7–2: Eliminate all unit productions from the following grammar:

$$\begin{array}{lcl} S & \rightarrow & 0S0 \mid A \\ A & \rightarrow & 101 \mid B \\ B & \rightarrow & C \mid 1 \\ C & \rightarrow & 1 \mid 111 \mid A \end{array}$$

Solution: Non-unit productions are:

$$\begin{array}{lcl} S & \rightarrow & 0S0 \\ A & \rightarrow & 101 \\ B & \rightarrow & 1 \\ C & \rightarrow & 1 \mid 111 \end{array}$$

By extending unit productions, we see that:
S can be replaced by A, B or C,
A can be replaced by B or C,
B can be replaced by A or C and
C can be replaced by A or B.

Eliminating all unit productions, we obtain:

$$\begin{array}{lcl} S & \rightarrow & 0S0 \mid 101 \mid 1 \mid 111 \\ A & \rightarrow & 101 \mid 1 \mid 111 \\ B & \rightarrow & 101 \mid 1 \mid 111 \\ C & \rightarrow & 101 \mid 1 \mid 111 \end{array}$$

Eliminating useless symbols, we get:

$$S \rightarrow 0S0 \mid 101 \mid 1 \mid 111$$

Epsilon Productions: We prefer not to have $\varepsilon-$productions in a grammar because they make the theoretical analysis of the length of intermediate strings very difficult.

If there are no $\varepsilon-$productions, the length must increase or stay the same at each step of the derivation. With ε, the length may go up and down.

Sometimes we cannot eliminate all $\varepsilon-$rules. If $\varepsilon \in L$, we have to keep $S \to \varepsilon$ but we can eliminate all the rest.

A variable A is **nullable** if we can obtain ε starting from A and using productions. For example, if the production $A \to \varepsilon$ exists, obviously A is nullable. On the other hand, if the productions

$$A \to \varepsilon, \quad B \to \varepsilon, \quad C \to AB$$

exist, C is nullable.

To eliminate $\varepsilon-$productions, we modify all the right-hand sides having nullable variables as follows: For each occurrence of a variable, add a rule (or rules) without it.

For example, $\begin{array}{lcl} S & \to & 0S0 \mid 1A1 \\ A & \to & 000 \mid \varepsilon \end{array}$ will be:

$$\begin{array}{lcl} S & \to & 0S0 \mid 1A1 \mid 11 \\ A & \to & 000 \end{array}$$

But, $\begin{array}{lcl} S & \to & 0S0 \mid 1A0A1 \\ A & \to & 000 \mid \varepsilon \end{array}$ will be:

$$\begin{array}{lcl} S & \to & 0S0 \mid 1A0A1 \mid 10A1 \mid 1A01 \mid 101 \\ A & \to & 000 \end{array}$$

Example 7–3: Eliminate rules containing ε from the following grammar:

$$\begin{array}{lcl} S & \rightarrow & 0S \mid B \mid 00C \\ A & \rightarrow & 111 \mid CC \mid \varepsilon \\ B & \rightarrow & A101C \mid 1 \\ C & \rightarrow & 11 \mid A \end{array}$$

Solution: The nullable variables are A and C.

$$\begin{array}{lcl} S & \rightarrow & 0S \mid B \mid 00C \mid 00 \\ A & \rightarrow & 111 \mid CC \mid C \\ B & \rightarrow & A101C \mid 101C \mid A101 \mid 101 \mid 1 \\ C & \rightarrow & 11 \mid A \end{array}$$

Example 7–4: Eliminate rules containing ε from the following grammar:

$$\begin{array}{lcl} S & \rightarrow & 0S \mid AB \\ A & \rightarrow & 1 \mid 11 \mid \varepsilon \\ B & \rightarrow & 0 \mid 00 \mid AAA \mid \varepsilon \end{array}$$

Solution: All three variables are nullable. It is impossible to eliminate $S \rightarrow \varepsilon$:

$$\begin{array}{lcl} S & \rightarrow & 0S \mid AB \mid A \mid B \mid \varepsilon \\ A & \rightarrow & 1 \mid 11 \\ B & \rightarrow & 0 \mid 00 \mid AAA \mid AA \mid A \end{array}$$

Using All Simplifications: We can simplify a grammar substantially by using the three kinds of simplifications together. But we have to be careful about the order.

A possible problem is that, while we are eliminating some kind of unwanted productions, we may be introducing new unwanted productions of another kind.

If we use the following order, we will eliminate all $\varepsilon-$productions, unit productions, and useless symbols:

1. Eliminate $\varepsilon-$productions.
2. Eliminate unit productions.
3. Eliminate useless symbols.

With a different order, the results are not guaranteed. For example,

$$\begin{array}{lcl} S & \rightarrow & 0A0 \\ A & \rightarrow & 1 \mid B \\ B & \rightarrow & 10 \mid 01 \end{array}$$

has no useless symbols. Eliminate unit productions:

$$\begin{array}{lcl} S & \rightarrow & 0A0 \\ A & \rightarrow & 1 \mid 10 \mid 01 \\ B & \rightarrow & 10 \mid 01 \end{array}$$

Now B is useless, because it is unreachable.

Similarly,

$$\begin{array}{lcl} S & \rightarrow & AB \\ A & \rightarrow & 0 \\ B & \rightarrow & 1 \mid \varepsilon \end{array}$$

has no unit productions. But removing $B \rightarrow \varepsilon$ creates $S \rightarrow A$.

Example 7–5: Eliminate all $\varepsilon-$productions, unit productions, and useless symbols from the following grammar:

$$\begin{aligned}
S &\rightarrow BAB \\
A &\rightarrow a \mid aA \\
B &\rightarrow bb \mid C \mid \varepsilon \\
C &\rightarrow aC \mid CC \mid DC \\
D &\rightarrow SA \mid AB
\end{aligned}$$

Solution: Eliminate $\varepsilon-$productions: $(B \rightarrow \varepsilon)$

$$\begin{aligned}
S &\rightarrow BAB \mid BA \mid AB \mid A \\
A &\rightarrow a \mid aA \\
B &\rightarrow bb \mid C \\
C &\rightarrow aC \mid CC \mid DC \\
D &\rightarrow SA \mid AB \mid A
\end{aligned}$$

Eliminate unit productions: $(S \rightarrow A,\ B \rightarrow C,\ D \rightarrow A)$

$$\begin{aligned}
S &\rightarrow BAB \mid BA \mid AB \mid a \mid aA \\
A &\rightarrow a \mid aA \\
B &\rightarrow bb \mid aC \mid CC \mid DC \\
C &\rightarrow aC \mid CC \mid DC \\
D &\rightarrow SA \mid AB \mid a \mid aA
\end{aligned}$$

Eliminate useless symbols: (C is non-generating, once we remove it, D becomes unreachable.)

$$\begin{aligned}
S &\rightarrow BAB \mid BA \mid AB \mid a \mid aA \\
A &\rightarrow a \mid aA \\
B &\rightarrow bb
\end{aligned}$$

Example 7–6: Eliminate all $\varepsilon-$productions, unit productions, and useless symbols from the following grammar:

$$\begin{aligned} S &\rightarrow AA \mid 0D0 \\ A &\rightarrow AC \\ B &\rightarrow 0 \mid 1 \\ C &\rightarrow 111 \mid AD \mid \varepsilon \\ D &\rightarrow 1 \mid C \end{aligned}$$

Solution: Nullable variables are C and D. Removing these:

$$\begin{aligned} S &\rightarrow AA \mid 0D0 \mid 00 \\ A &\rightarrow AC \\ B &\rightarrow 0 \mid 1 \\ C &\rightarrow 111 \mid AD \mid A \\ D &\rightarrow 1 \mid C \end{aligned}$$

Now remove unit productions:
($D \rightarrow C,\ C \rightarrow A$ and $D \rightarrow A$)

$$\begin{aligned} S &\rightarrow AA \mid 0D0 \mid 00 \\ A &\rightarrow AC \\ B &\rightarrow 0 \mid 1 \\ C &\rightarrow 111 \mid AD \mid AC \\ D &\rightarrow 1 \mid 111 \mid AD \mid AC \end{aligned}$$

B is unreachable. A is non-generating. If we remove A, C becomes unreachable. Remove them:

$$\begin{aligned} S &\rightarrow 0D0 \mid 00 \\ D &\rightarrow 1 \mid 111 \end{aligned}$$

7.2 Chomsky Normal Form

A context-free grammar is in **Chomsky Normal Form (CNF)** if every production is of the form

$$\begin{aligned} A &\to BC \\ A &\to a \end{aligned}$$

where a denotes a terminal and A, B, C denote variables where neither B nor C is the start variable.

In addition, there is the production

$$S \to \varepsilon$$

if and only if ε belongs to the language.

For example, the following grammars over the alphabet $\Sigma = \{a, b, c\}$ are in CNF:

$$\begin{aligned} S &\to AB \mid AA \\ A &\to a \\ B &\to BC \mid CC \mid b \\ C &\to a \mid c \end{aligned} \qquad \begin{aligned} S &\to TU \mid a \mid \varepsilon \\ T &\to UU \mid TU \mid UT \\ U &\to TT \mid a \mid b \mid c \end{aligned}$$

$$\begin{aligned} S &\to AB \mid AC \mid BC \\ A &\to AA \mid a \\ B &\to BB \mid b \\ C &\to a \mid b \mid c \end{aligned} \qquad \begin{aligned} S &\to TU \mid BT \mid \varepsilon \\ T &\to AB \mid AT \mid c \\ U &\to BB \mid BU \mid c \\ A &\to a \\ B &\to b \end{aligned}$$

The following grammars are NOT in CNF:

$$\begin{array}{lcl} S & \to & ABA \mid AA \\ A & \to & a \\ B & \to & b \end{array} \qquad \begin{array}{lcl} S & \to & AB \mid B \\ A & \to & aA \\ B & \to & b \end{array}$$

$$\begin{array}{lcl} S & \to & TT \mid a \\ T & \to & ST \mid b \end{array} \qquad \begin{array}{lcl} S & \to & TT \mid a \\ T & \to & a \mid b \mid \varepsilon \end{array}$$

The advantage of CNF is that any derivation of a string of length n takes exactly $2n-1$ steps. For example, derivation of abc takes exactly 5 steps:

$$S \to AT \to ABC \to aBC \to abC \to abc$$

This helps a lot in proving certain facts about CFLs and other theoretical matters. Furthermore, we can modify any given grammar and obtain another one in CNF.

Theorem: Any context-free language is generated by a context-free grammar in Chomsky Normal Form.

Sketch of Proof:

1. Eliminate all $\varepsilon-$productions and unit productions. (It is optional to eliminate useless symbols.) Also, add a new start variable if necessary.

2. Eliminate terminals on the right-hand side except for productions of the form $A \to a$. If necessary, add new variables and productions.

3. Eliminate right-hand sides containing more than 2 variables by defining new variables and productions.

Example 7–7: Convert the following CFG to CNF:

$$\begin{aligned} S &\rightarrow 0A0 \mid 1B \\ A &\rightarrow 1 \mid BB \\ B &\rightarrow 111 \mid A0 \end{aligned}$$

Solution: There are no $\varepsilon-$productions, unit productions, or useless symbols in this grammar. Also, the start variable does not appear on the right-hand side. So we do not need a new start variable.

We need to eliminate productions involving terminals except for $A \rightarrow 1$. We can easily do this by introducing two new variables, W and Z:

$$\begin{aligned} S &\rightarrow ZAZ \mid WB \\ A &\rightarrow 1 \mid BB \\ B &\rightarrow WWW \mid AZ \\ W &\rightarrow 1 \\ Z &\rightarrow 0 \end{aligned}$$

Now, we have to reduce right-hand sides involving 3 variables:

$$\begin{aligned} S &\rightarrow ZC \mid WB \\ A &\rightarrow 1 \mid BB \\ B &\rightarrow WD \mid AZ \\ C &\rightarrow AZ \\ D &\rightarrow WW \\ W &\rightarrow 1 \\ Z &\rightarrow 0 \end{aligned}$$

This grammar is in Chomsky Normal Form.

Example 7–8: Convert the following grammar into CNF:

$$\begin{aligned} S &\rightarrow A1A \\ A &\rightarrow 0B0 \mid \varepsilon \\ B &\rightarrow A \mid 10 \end{aligned}$$

Solution: Eliminate $\varepsilon-$ and unit productions:

$$\begin{aligned} S &\rightarrow A1A \mid 1A \mid A1 \mid 1 \\ A &\rightarrow 0B0 \mid 0A0 \mid 00 \\ B &\rightarrow 10 \end{aligned}$$

Eliminate terminals (which are not single):

$$\begin{aligned} S &\rightarrow AWA \mid WA \mid AW \mid 1 \\ A &\rightarrow ZBZ \mid ZAZ \mid ZZ \\ B &\rightarrow WZ \\ W &\rightarrow 1 \\ Z &\rightarrow 0 \end{aligned}$$

Break variable strings longer than 2:

$$\begin{aligned} S &\rightarrow AC \mid WA \mid AW \mid 1 \\ A &\rightarrow ZD \mid ZE \mid ZZ \\ B &\rightarrow WZ \\ C &\rightarrow WA \\ D &\rightarrow BZ \\ E &\rightarrow AZ \\ W &\rightarrow 1 \\ Z &\rightarrow 0 \end{aligned}$$

Example 7–9: Find a CFG in CNF that generates the language

$$L = \{w \mid |w| = 8k, \quad k = 1, 2, 3, \dots\}$$

over the alphabet $\Sigma = \{a, b\}$.

Solution: A simple grammar that generates L is:

$$\begin{aligned} S &\rightarrow A \mid AS \\ A &\rightarrow DDDDDDDD \\ D &\rightarrow a \mid b \end{aligned}$$

Using the first rule, we will reach a string that contains one or more A's, in other words, $A^k,\ k \geqslant 1$.

Using the second rule will give us $D^{8k},\ k \geqslant 1$ and then the third rule finishes the job.

This grammar is not in Chomsky Normal Form, but we can easily transform it. First eliminate the unit rule:

$$\begin{aligned} S &\rightarrow DDDDDDDD \mid AS \\ A &\rightarrow DDDDDDDD \\ D &\rightarrow a \mid b \end{aligned}$$

Now we have to break the D^8 part, but we do not need to use 7 new variables. Instead, we can do the following trick:

$$\begin{aligned} S &\rightarrow AA \mid BB \\ A &\rightarrow AA \mid BB \\ B &\rightarrow CC \\ C &\rightarrow DD \\ D &\rightarrow a \mid b \end{aligned}$$

7.3 Regular Grammars

A grammar where all the productions have the form

$$\begin{aligned} A &\rightarrow aB, \quad \text{or} \\ A &\rightarrow a \end{aligned}$$

is called a **Regular Grammar**. Here A and B denote variables and a denotes a terminal. Note that $A \rightarrow \varepsilon$ is also allowed. For example,

$$\begin{aligned} S &\rightarrow 1A \mid \varepsilon \\ A &\rightarrow 0S \end{aligned}$$

is a regular grammar. It generates the language $(10)^*$.

Regular grammars are among the simplest of all grammars. Also, they have a connection with regular languages:

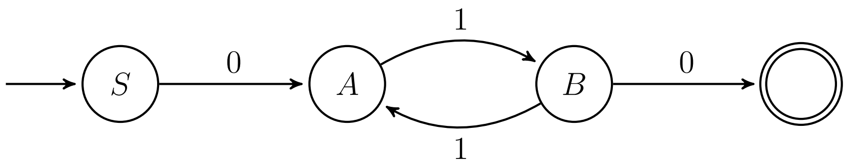
Theorem: A language is regular if and only if it can be generated by a regular grammar.

Sketch of Proof: The basic idea is to think of each variable as a state in an NFA. We also need an extra (and unique) accept state. The transitions $A \rightarrow a$ and $A \rightarrow \varepsilon$ will take us to the accept state.

For example,

$$S \rightarrow 0A, \quad A \rightarrow 1B, \quad B \rightarrow 0 \mid 1A$$

is equivalent to:



Example 7–10: Find a regular grammar that generates the language

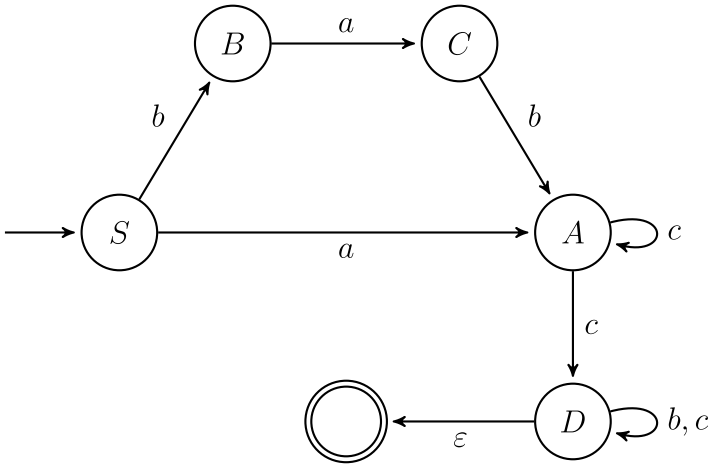
$$(a \cup bab)c^+(b \cup c)^*$$

over the alphabet $\Sigma = \{a, b, c\}$.

Solution: Using the given regular expression, we can construct the grammar directly:

$$\begin{array}{rcl} S & \to & aA \mid bB \\ A & \to & cA \mid cD \\ B & \to & aC \\ C & \to & bA \\ D & \to & bD \mid cD \mid \varepsilon \end{array}$$

Or we can first construct the following NFA using the regular expression:



Then, we can obtain the grammar using this machine.

EXERCISES

Eliminate all useless symbols and productions from the following grammars:

7–1)
$$\begin{aligned} S &\to 0S00 \mid 1 \mid A \\ A &\to 1B11 \mid BB \\ B &\to 0B0 \mid BCB \\ C &\to 00 \mid 1S1 \end{aligned}$$

7–2)
$$\begin{aligned} S &\to aaS \mid C \mid D \\ A &\to aSbb \mid B \\ B &\to bB \mid aCa \mid \varepsilon \\ C &\to b \mid bb \mid bbb \\ D &\to aDa \mid CBD \end{aligned}$$

Eliminate unit productions from the following grammars:

7–3)
$$\begin{aligned} S &\to 0 \mid A \mid 11C \\ A &\to 10 \mid 01 \mid B \\ B &\to 0110 \\ C &\to 00 \mid B \end{aligned}$$

7–4)
$$\begin{aligned} S &\to SS \mid AB \\ A &\to aA \mid C \mid \varepsilon \\ B &\to bBb \mid C \mid bb \\ C &\to ABC \mid bb \end{aligned}$$

Eliminate $\varepsilon-$productions from the following grammars:

7–5)
$$\begin{aligned} S &\rightarrow aA \mid BaB \\ A &\rightarrow aaa \mid B \\ B &\rightarrow bB \mid \varepsilon \end{aligned}$$

7–6)
$$\begin{aligned} S &\rightarrow SA \mid A11B1 \\ A &\rightarrow 000 \mid \varepsilon \\ B &\rightarrow 00 \mid \varepsilon \end{aligned}$$

Eliminate all useless, unit and $\varepsilon-$productions from the following grammars:

7–7)
$$\begin{aligned} S &\rightarrow aaA \mid BC \mid aDaD \\ A &\rightarrow b \mid B \mid \varepsilon \\ B &\rightarrow b \mid bab \\ C &\rightarrow aC \mid CC \\ D &\rightarrow a \mid A \mid C \end{aligned}$$

7–8)
$$\begin{aligned} S &\rightarrow 01S \mid A \\ A &\rightarrow 01 \mid BC \\ B &\rightarrow 010 \\ C &\rightarrow CC \mid \varepsilon \\ D &\rightarrow 0C1C \mid \varepsilon \end{aligned}$$

Convert the following CFGs to Chomsky normal form.

7–9)
$$\begin{aligned} S &\rightarrow A \mid AB \mid ABAA \\ A &\rightarrow 0 \\ B &\rightarrow 1 \end{aligned}$$

7–10)
$$\begin{aligned} S &\rightarrow AA \mid AB \\ A &\rightarrow 0 \mid \varepsilon \\ B &\rightarrow 1 \mid \varepsilon \end{aligned}$$

7–11)
$$\begin{aligned} S &\rightarrow SA \mid 1 \\ A &\rightarrow 0A1 \mid S \mid 0 \end{aligned}$$

7–12)
$$\begin{aligned} S &\rightarrow aC \mid CDC \\ C &\rightarrow bb \\ D &\rightarrow bD \mid Da \mid \varepsilon \end{aligned}$$

7–13)
$$\begin{aligned} S &\rightarrow CaC \\ C &\rightarrow aDa \mid \varepsilon \\ D &\rightarrow C \mid bb \end{aligned}$$

7–14)
$$\begin{aligned} S &\rightarrow aaa \mid DC \\ C &\rightarrow CD \mid \varepsilon \\ D &\rightarrow bb \mid bbb \mid \varepsilon \end{aligned}$$

7–15) Find a CFG over $\Sigma = \{0, 1\}$ that generates strings containing an odd number of symbols, starting and ending with 1. Then, express the same grammar in Chomsky normal form.

ANSWERS

7–1) $S \rightarrow 0S00 \mid 1$

7–2)
$$\begin{aligned} S &\rightarrow aaS \mid C \\ C &\rightarrow b \mid bb \mid bbb \end{aligned}$$

7–3)
$$\begin{aligned} S &\rightarrow 0 \mid 10 \mid 01 \mid 0110 \mid 11C \\ C &\rightarrow 00 \mid 0110 \end{aligned}$$

7–4)
$$\begin{aligned} S &\rightarrow SS \mid AB \\ A &\rightarrow aA \mid ABC \mid bb \mid \varepsilon \\ B &\rightarrow bBb \mid ABC \mid bb \\ C &\rightarrow ABC \mid bb \end{aligned}$$

7–5)
$$\begin{aligned} S &\rightarrow aA \mid BaB \mid a \mid aB \mid Ba \\ A &\rightarrow aaa \mid B \\ B &\rightarrow bB \mid b \end{aligned}$$

7–6)
$$\begin{aligned} S &\rightarrow SA \mid A11B1 \mid 11B1 \mid A111 \mid 111 \\ A &\rightarrow 000 \\ B &\rightarrow 00 \end{aligned}$$

7–7)
$$\begin{aligned} S &\rightarrow aa \mid aaA \mid aaD \mid aDa \mid aDaD \\ A &\rightarrow b \mid bab \\ D &\rightarrow a \mid b \mid bab \end{aligned}$$

7–8) $S \rightarrow 01S \mid 01 \mid 010$

7–9)
$$\begin{aligned} S &\rightarrow 0 \mid AB \mid CD \\ C &\rightarrow AB \\ D &\rightarrow AA \\ A &\rightarrow 0 \\ B &\rightarrow 1 \end{aligned}$$

7–10)
$$\begin{aligned} S &\rightarrow AA \mid AB \mid 0 \mid 1 \mid \varepsilon \\ A &\rightarrow 0 \\ B &\rightarrow 1 \end{aligned}$$

7–11)
$$\begin{aligned} S_0 &\rightarrow SA \mid 1 \\ S &\rightarrow SA \mid 1 \\ A &\rightarrow CB \mid SA \mid 1 \mid 0 \\ B &\rightarrow AD \\ C &\rightarrow 0 \\ D &\rightarrow 1 \end{aligned}$$

7–12)
$$\begin{aligned} S &\rightarrow AC \mid CC \mid CE \\ E &\rightarrow DC \\ C &\rightarrow BB \\ D &\rightarrow BD \mid DA \mid b \mid a \\ A &\rightarrow a \\ B &\rightarrow b \end{aligned}$$

7–13)

$$\begin{aligned}
S &\rightarrow CE \mid AC \mid CA \mid a \\
C &\rightarrow AF \mid AA \\
D &\rightarrow AF \mid AA \mid BB \\
E &\rightarrow AC \\
F &\rightarrow DA \\
A &\rightarrow a \\
B &\rightarrow b
\end{aligned}$$

7–14)

$$\begin{aligned}
S &\rightarrow AE \mid DC \mid CD \mid BB \mid BF \mid \varepsilon \\
C &\rightarrow CD \mid BB \mid BF \\
D &\rightarrow BB \mid BF \\
E &\rightarrow AA \\
F &\rightarrow BB \\
A &\rightarrow a \\
B &\rightarrow b
\end{aligned}$$

7–15)

$$\begin{aligned}
S &\rightarrow 1A1 \mid 1 \\
A &\rightarrow BAB \mid B \\
B &\rightarrow 0 \mid 1
\end{aligned}$$

Modified grammar in CNF:

$$\begin{aligned}
S &\rightarrow CW \mid 1 \\
C &\rightarrow WA \\
W &\rightarrow 1 \\
A &\rightarrow BD \mid 0 \mid 1 \\
D &\rightarrow AB \\
B &\rightarrow 0 \mid 1
\end{aligned}$$

Week 8

Pushdown Automata

8.1 Operation of a PDA

We have seen that NFAs have a very limited capacity to keep things in mind, in other words, memory. Therefore they cannot recognize many languages, for example, $a^n\, b^n$. Now we will add memory to an NFA and obtain a more powerful machine.

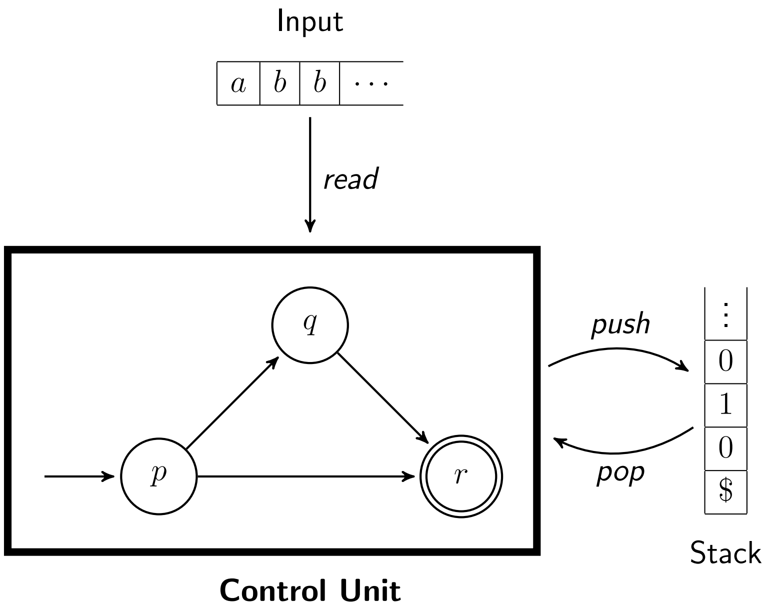
A **stack** is a LIFO (Last In, First Out) memory. Writing a symbol is called **pushing**.

We can only read the symbol on the top of the stack. Once we read it, it is removed. This is called **popping**.

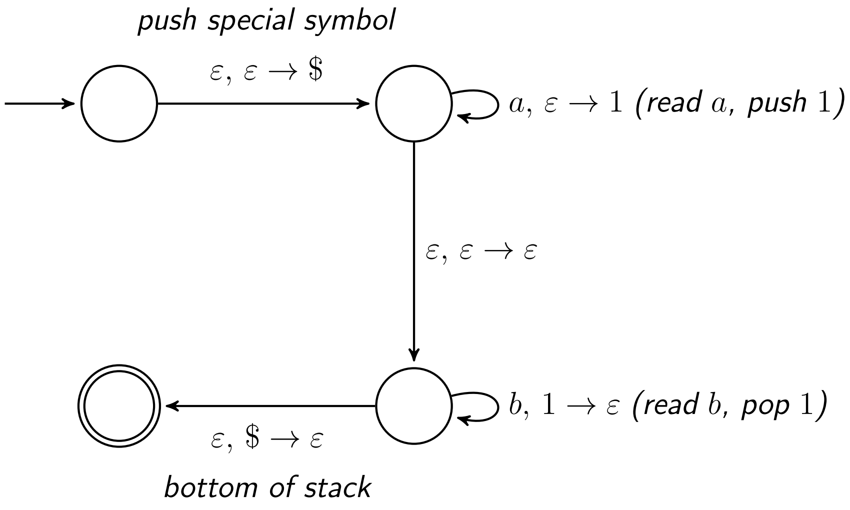
A pushdown automaton is, informally, an NFA plus a stack. The stack is assumed to have unlimited size.

Using a special symbol $\$$ in the beginning of computation, we can test whether we are at the bottom of the stack or not.

The transition function for a PDA is more complicated than that of an NFA because PDA chooses the next state based on the current state, the input, *and* the stack. As usual, we can use the ε symbol.



For example, the PDA that recognizes $\{a^n b^n \mid n \geqslant 0\}$ is:


push special symbol
$\varepsilon,\ \varepsilon \to \$$
$a,\ \varepsilon \to 1$ (read a, push 1)
$\varepsilon,\ \varepsilon \to \varepsilon$
$b,\ 1 \to \varepsilon$ (read b, pop 1)
$\varepsilon,\ \$ \to \varepsilon$
bottom of stack

Formal Definition of a PDA:

A **Pushdown Automaton (PDA)** is a $6-$tuple

$$\big(Q, \Sigma, \Gamma, \delta, q_0, F\big)$$

where Q, Σ, Γ and F are all finite sets, and

- Q is the set of states,
- Σ is the input alphabet,
- Γ is the stack alphabet,
- δ is the transition function,
  $\delta : Q \times \big(\Sigma \cup \varepsilon\big) \times \big(\Gamma \cup \varepsilon\big) \rightarrow$ all subsets of $Q \times \big(\Gamma \cup \varepsilon\big)$.
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ is the set of accept states.

Note that Σ and Γ may or may not be the same. We will usually use different alphabets for input and stack to make things clear. For example, suppose:

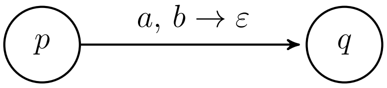
$$\begin{aligned} Q &= \{q_1, q_2, q_3\} \\ \Sigma &= \{a, b, c\} \\ \Gamma &= \{\$, 0, 1\} \end{aligned}$$

Then, the notation $\delta\big(q_1, a, 1\big) = \big\{\big(q_2, 0\big), \big(q_3, \varepsilon\big)\big\}$ means:

If you are in state q_1, read a as input and pop 1 from the stack, go to state q_2 and push 0 to stack, or go to state q_3 and push nothing to stack.

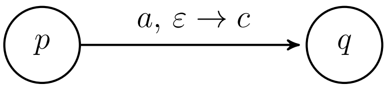
Here are some examples of possible moves and how we denote them in state diagrams of a PDA:

- Read input a, pop b from stack, move from state p to q: (Do not push anything to stack.)

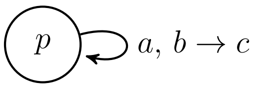
  

  (Note that, if the input at that stage is not a or if the symbol on top of the stack is not b, this branch of computation dies.)

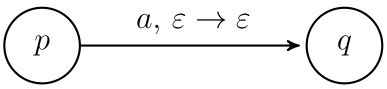
- Read input a, push c to stack, move from state p to q: (Do not read from the stack. Do not pop anything.)

- Read input a, pop b from stack, push c to stack, stay in the same state:

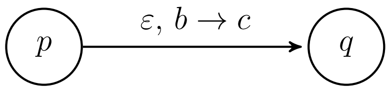
  

- Read input a, move from state p to q:

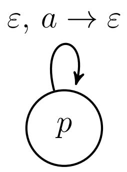
  

  (Do not push or pop anything. This move is just like an NFA.)

- Do not read input. Pop b from stack, push c to stack, move from state p to q:



- Do not read input. Pop a from stack. Stay in the same state:

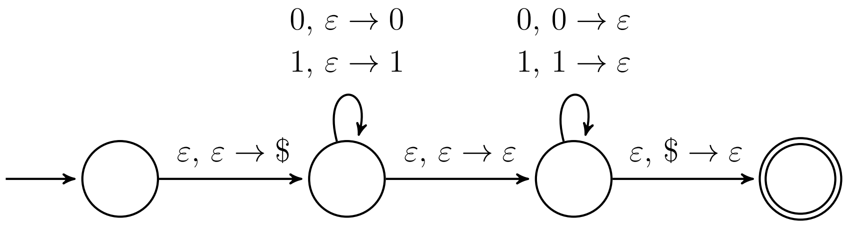


Example 8–1: Find a PDA that recognizes the language

$$L = \left\{ ww^R \,\middle|\, w \in \{0,1\}^* \right\}$$

where w^R indicates the reverse of the string w.

Solution: This is the language of even palindromes. Up to a point, we push symbols to stack and after that, we pop them. Order is reversed because of the way a stack works.

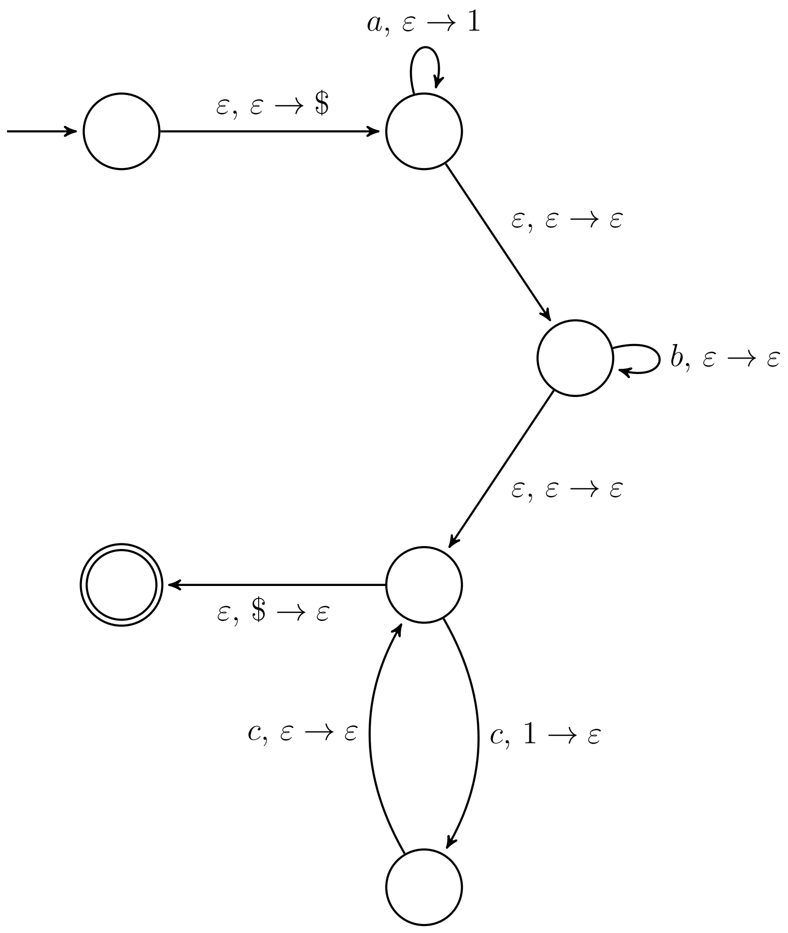

$0,\, \varepsilon \to 0$
$1,\, \varepsilon \to 1$
$0,\, 0 \to \varepsilon$
$1,\, 1 \to \varepsilon$
$\varepsilon,\, \varepsilon \to \$$
$\varepsilon,\, \varepsilon \to \varepsilon$
$\varepsilon,\, \$ \to \varepsilon$

But how do we know we are in the middle of the string? We guess using nondeterminacy!

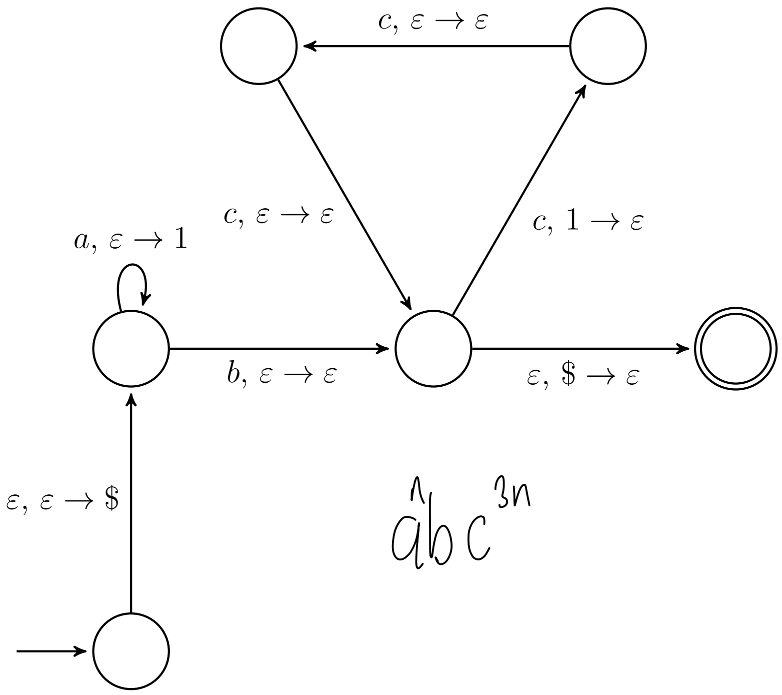
Example 8–2: Construct a PDA that recognizes the language

$$L = \left\{a^n b^m c^{2n} \;\middle|\; m, n \geqslant 0\right\}$$

Solution: We have to count the a's. There is an arbitrary number of b's, maybe none. We do not have to count them. For every two c's in the input, we should decrease count by one.


$a,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \varepsilon \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$
$c,\ \varepsilon \to \varepsilon$
$c,\ 1 \to \varepsilon$

Example 8–3: What language does the following PDA recognize?


$c,\ \varepsilon \to \varepsilon$
$a,\ \varepsilon \to 1$
$c,\ \varepsilon \to \varepsilon$
$c,\ 1 \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$
$\varepsilon,\ \varepsilon \to \$$

$\hat{a} b c^{3n}$

Solution: At first, we push $\$$ to stack. Then, for each a, we push 1. Then we read a single b from the input and push or pop nothing.

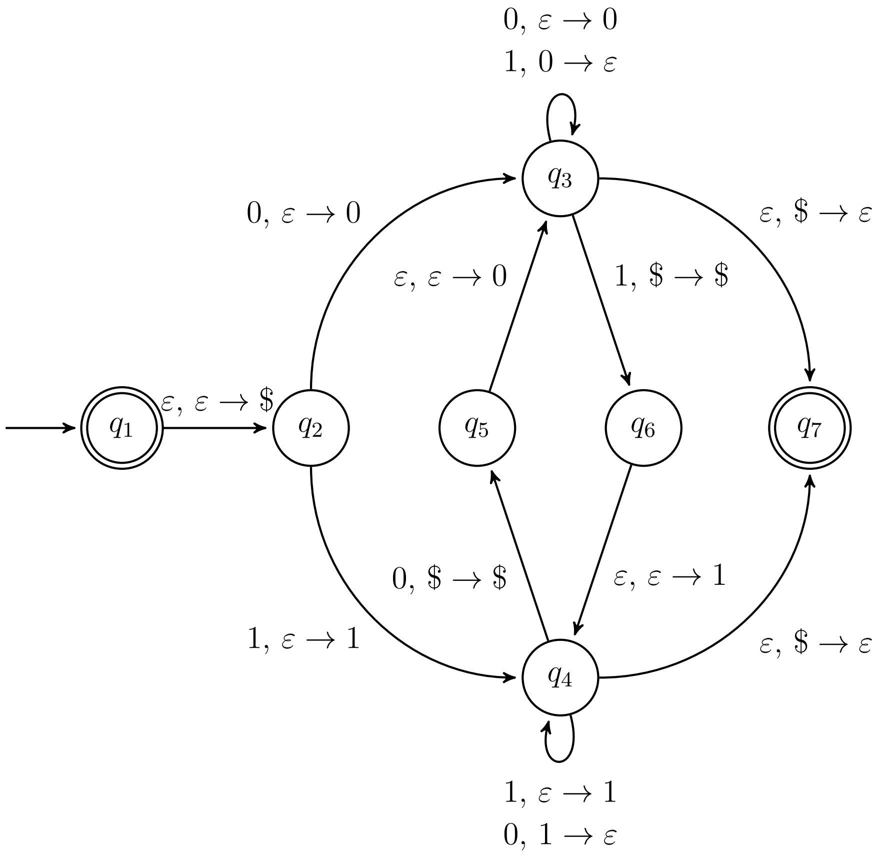
Now we are on the loop. We pop 1 from the stack for every three c's we read from input. If we meet the bottom of the stack, $\$$, we accept.

This description is equivalent to:

$$a^n\, b\, c^{3n}, \quad n \geqslant 0$$

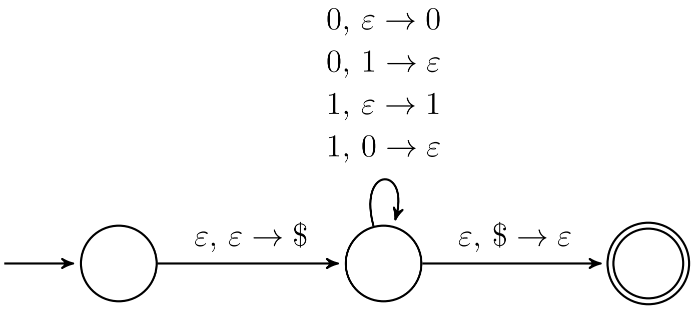
Example 8–4: Construct a PDA for the language of strings that contain equal numbers of 0's and 1's over $\Sigma = \{0, 1\}$.

Solution: We will push 1's to stack and pop them for each 0. But what if there are temporarily more 0's? Then, we have to push 0's and pop them as we receive 1's.



State q_3: Up to now, 0's $\geqslant$ 1's.
State q_4: Up to now, 1's $\geqslant$ 0's.

A better design is as follows:



Here, we are taking advantage of nondeterminacy.

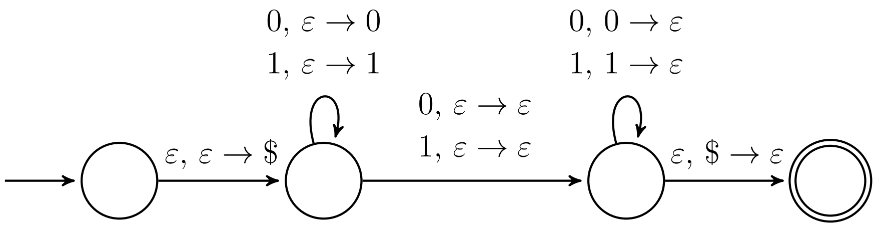
Example 8–5: Construct a PDA that recognizes over $\Sigma = \{0, 1\}$:

a) Palindromes of odd length.

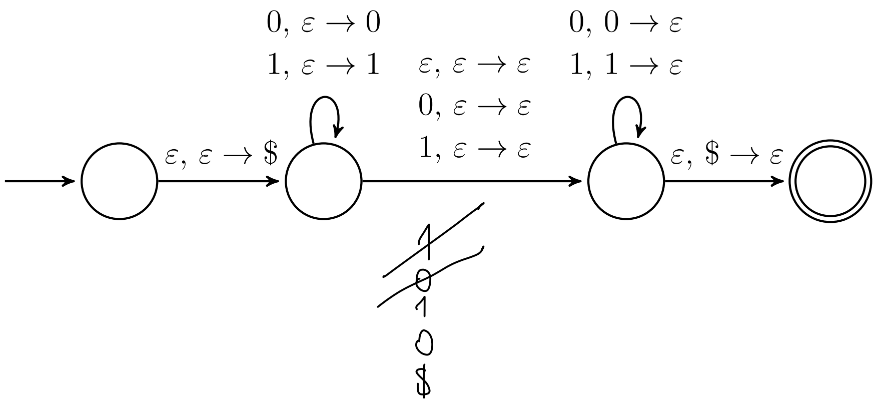
b) All palindromes.

0101|0|1001

Solution:



$0,\ \varepsilon \to 0$
$1,\ \varepsilon \to 1$
$0,\ 0 \to \varepsilon$
$1,\ 1 \to \varepsilon$
$0,\ \varepsilon \to \varepsilon$
$1,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \$ \to \varepsilon$

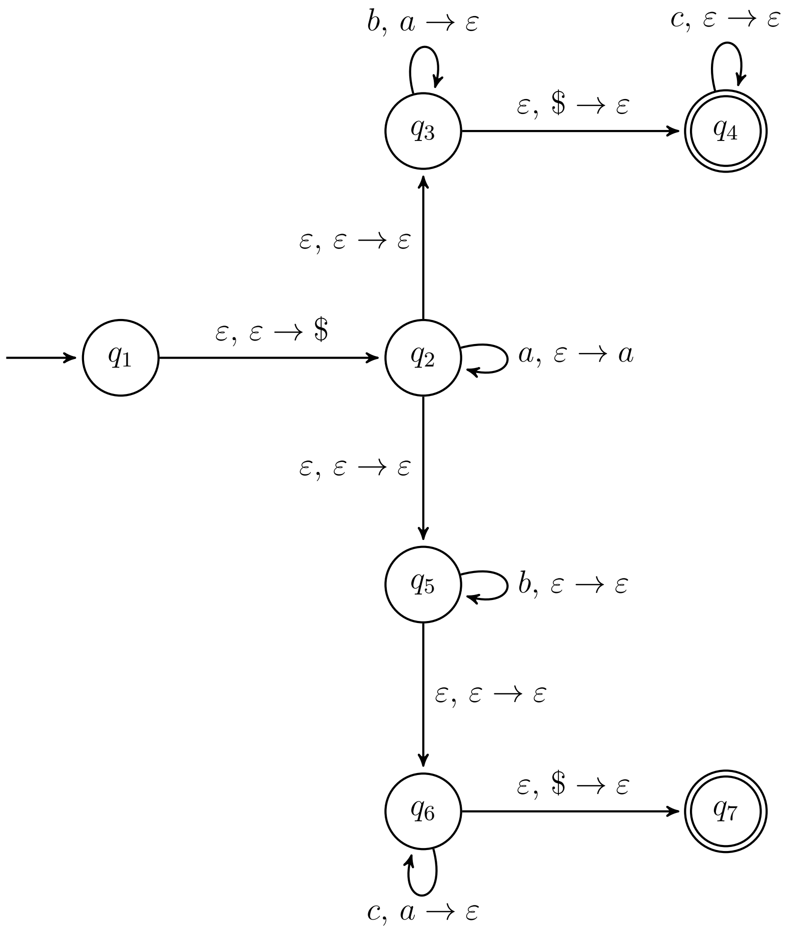


$0,\ \varepsilon \to 0$
$1,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to \varepsilon$
$0,\ \varepsilon \to \varepsilon$
$1,\ \varepsilon \to \varepsilon$
$0,\ 0 \to \varepsilon$
$1,\ 1 \to \varepsilon$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \$ \to \varepsilon$
1
0
1
0
$

Example 8–6: Find a PDA that recognizes the language:

$$L = \left\{ a^k\, b^m\, c^n \;\middle|\; k, m, n \geqslant 0 \text{ and } \big(k = m \text{ or } k = n\big) \right\}$$

Solution: Again we need nondeterminacy. Basically, we connect two machines in parallel such that each one checks one of these alternative conditions.



q_1
q_2
q_3
q_4
q_5
q_6
q_7
$\varepsilon,\ \varepsilon \to \$$
$a,\ \varepsilon \to a$
$\varepsilon,\ \varepsilon \to \varepsilon$
$b,\ a \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$
$c,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
$c,\ a \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$

8.2 PDA - CFG equivalence

Previously, we have seen that regular expressions and NFAs are two different ways of expressing the same languages. There is the same relationship between CFG and PDA:

Theorem: A language is context-free if and only if there is a PDA recognizing it.

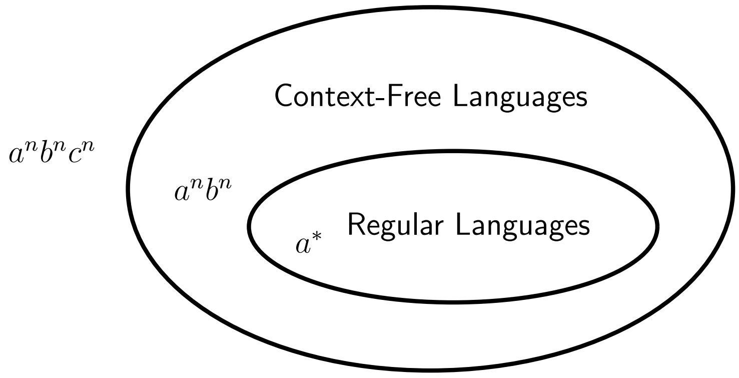
This theorem has two parts:

- Given a CFG, we can find an equivalent PDA.
- Given a PDA, we can find an equivalent CFG.

We omit the proof. We will return to this theorem again in the next chapter.

Theorem: Every regular language is context-free.

We have already seen that regular grammars are a simple type of CFG and they generate regular languages. But we can prove this theorem using the previous one more easily. Every NFA is a PDA that doesn't use its stack therefore any language recognized by an NFA is a CFL.



Here are some examples illustrating the idea that if we can find a PDA for a language, then we can also find a CFG for the same language and vice versa: (Keep in mind that giving examples is NOT equivalent to a proof.)

Example 8–7: Are the following languages context-free? If your answer is "Yes", find both a PDA and a CFG for that language. If your answer is "No", explain why not.

a) $\{a^n b^m c^m d^n \mid n, m \geqslant 0\}$

b) $\{a^n b^m c^n d^m \mid n, m \geqslant 0\}$

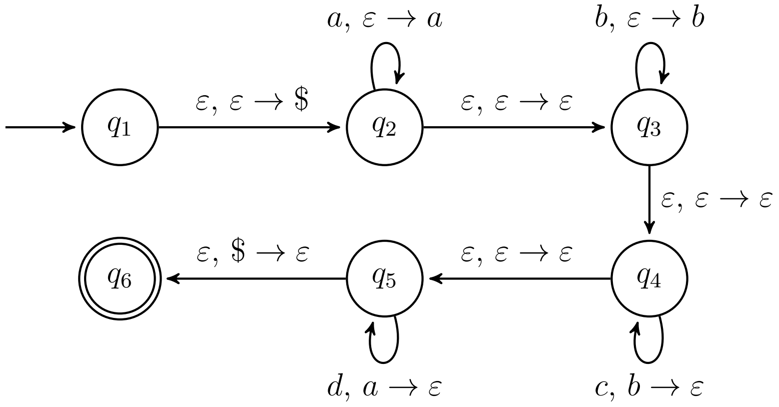
c) $\{a^n b^n c^m d^m \mid n, m \geqslant 0\}$

d) $\{a^n b^n c^n \mid n \geqslant 0\}$

e) $\{a^k b^m c^n \mid k > m > n\}$

Solution: a) Yes.

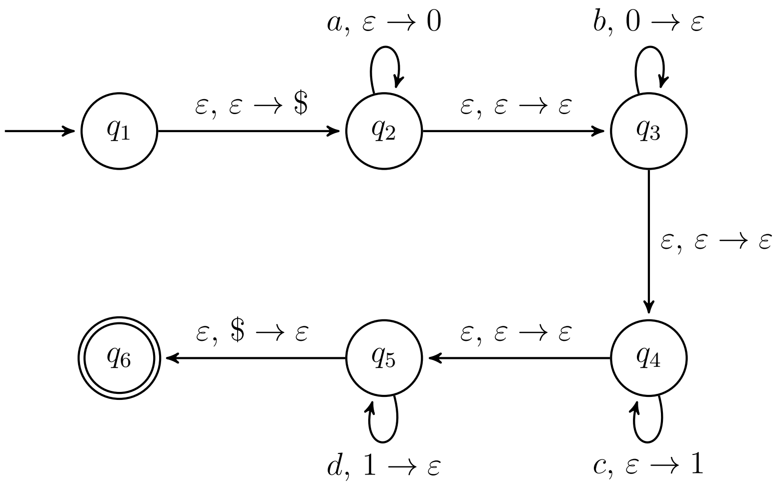
$$\begin{aligned} S &\rightarrow aSd \mid T \\ T &\rightarrow bTc \mid \varepsilon \end{aligned}$$


$a,\ \varepsilon \to a$
$b,\ \varepsilon \to b$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \varepsilon \to \varepsilon$
q_1
q_2
q_3
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
q_6
q_5
q_4
$d,\ a \to \varepsilon$
$c,\ b \to \varepsilon$

b) No. We can put a's and then b's to stack, but when we receive c's, we cannot reach a's.

c) Yes.

$$\begin{array}{rcl} S & \to & AC \\ A & \to & aAb \mid \varepsilon \\ C & \to & cCd \mid \varepsilon \end{array}$$


$a,\ \varepsilon \to 0$
$b,\ 0 \to \varepsilon$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \varepsilon \to \varepsilon$
q_1
q_2
q_3
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
q_6
q_5
q_4
$d,\ 1 \to \varepsilon$
$c,\ \varepsilon \to 1$

d) No. We can store the number of a's and then compare it with the number of b's. But when we receive c's, the stack is empty.

e) No. For the same reason. One stack is not sufficient for making comparisons of a to b and then b to c.

Note that our explanations for parts b), d), e) are quite informal and all about PDAs. We will see a formal method for proving that a language is not context-free in the next chapter.

Example 8–8: Is the following language context-free?

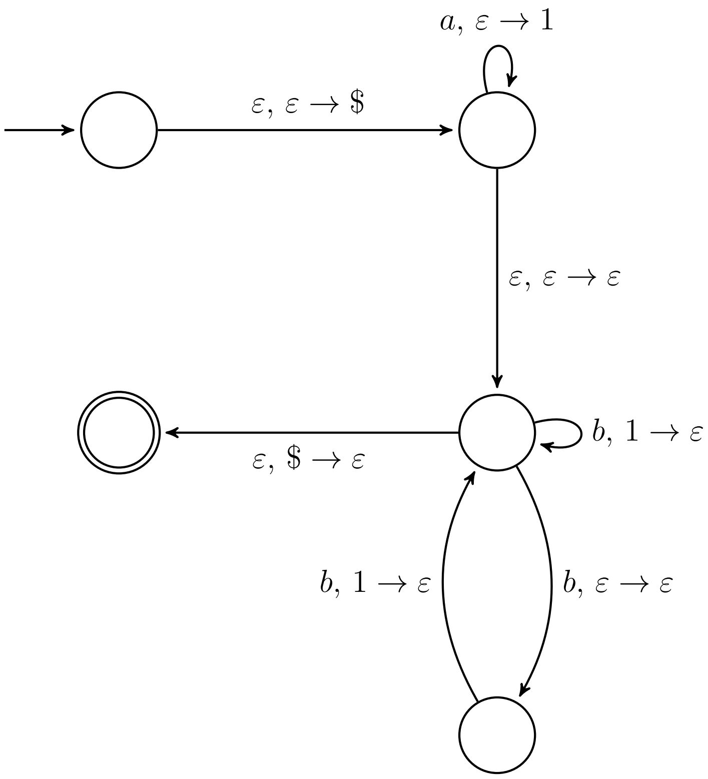
$$L = \left\{ a^n \, b^m \;\middle|\; n \leqslant m \leqslant 2n \right\}$$

Solution: Yes. We can show that in two different ways.

A CFG is:

$$S \;\; \rightarrow \;\; aSb \;\big|\; aSbb \;\big|\; \varepsilon$$

A PDA is:



Example 8–9: Is the following language context-free?

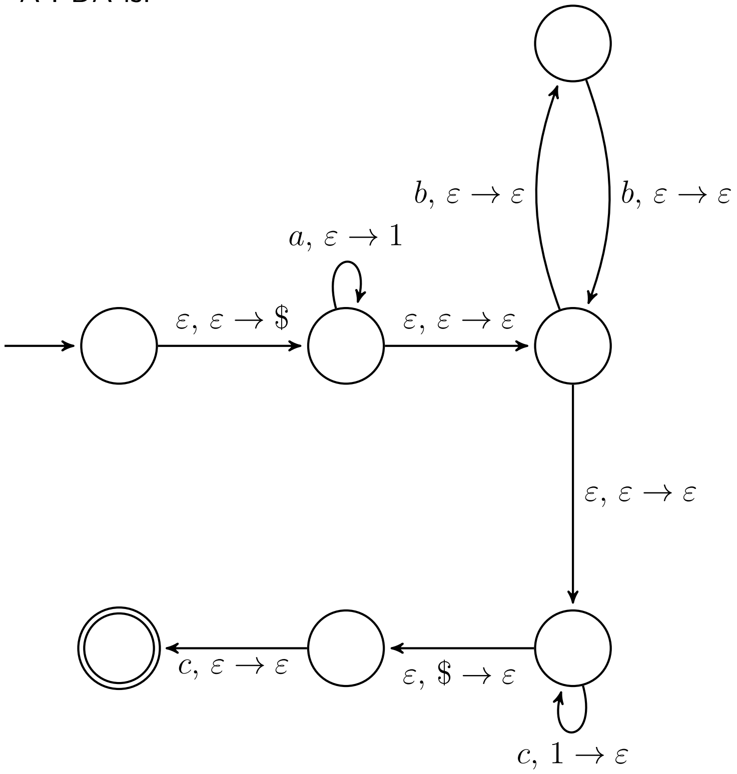
$$L = \left\{ a^n \, b^{2m} \, c^{n+1} \;\middle|\; n, m \geqslant 0 \right\}$$

Solution: Yes. We can show that in two different ways.

A CFG is:

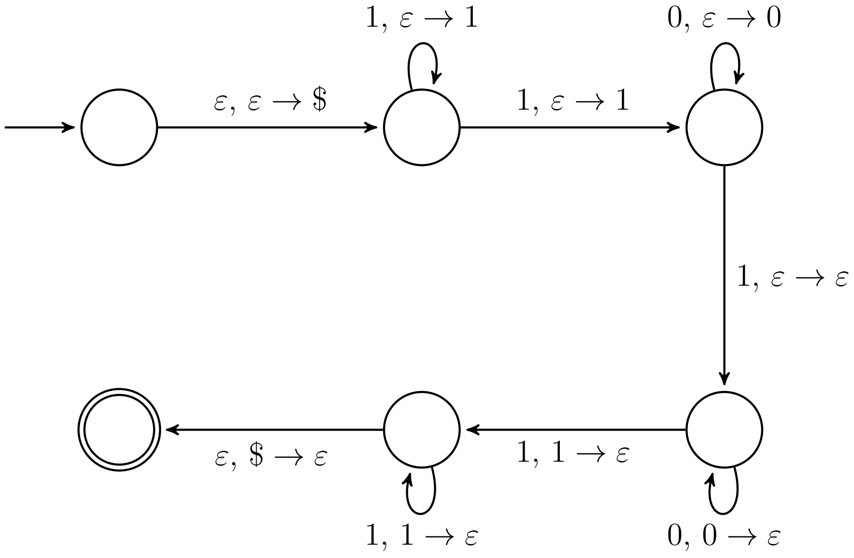
$$\begin{aligned} S &\rightarrow aSc \mid T \\ T &\rightarrow bbT \mid c \end{aligned}$$

A PDA is:


$\varepsilon,\, \varepsilon \rightarrow \$$
$a,\, \varepsilon \rightarrow 1$
$\varepsilon,\, \varepsilon \rightarrow \varepsilon$
$b,\, \varepsilon \rightarrow \varepsilon$
$b,\, \varepsilon \rightarrow \varepsilon$
$\varepsilon,\, \varepsilon \rightarrow \varepsilon$
$c,\, 1 \rightarrow \varepsilon$
$\varepsilon,\, \$ \rightarrow \varepsilon$
$c,\, \varepsilon \rightarrow \varepsilon$

$1^n 1 0^n 1 0^n 1 1^n$

Example 8–10: Consider the following PDA:



a) Describe the language recognized by the PDA.

b) Find a CFG generating the same language.

Solution: Following the arrows, we see that there is at least one 1, then any number of 0's, then a single 1 and the rest is symmetric. The language is:

$$L = \left\{1^{n+1}0^m10^m1^{n+1} \;\middle|\; n, m \geqslant 0\right\}$$

A CFG for this language is:

$$\begin{aligned} S &\to 1S1 \mid 1A1 \\ A &\to 0A0 \mid 1 \end{aligned}$$

EXERCISES

Construct PDAs recognizing the following languages:

8–1) $L = \{a^n\, b^{2n} \mid n \geqslant 0\}$

8–2) $L = \{a^{2n}\, b^n \mid n \geqslant 0\}$

8–3) $L = \{a^n\, b^2\, c^n \mid n \geqslant 0\}$

8–4) $L = \{a^n\, b^m\, c^n \mid n, m \geqslant 0\}$

8–5) $L = \{a^n\, b^m \mid n \neq m\}$

8–6) $L = \{w \mid \text{length of } w \text{ is odd and its middle symbol is } a\}$

8–7) $L = \{a^n\, b^{2n-1} \mid n \geqslant 1\}$

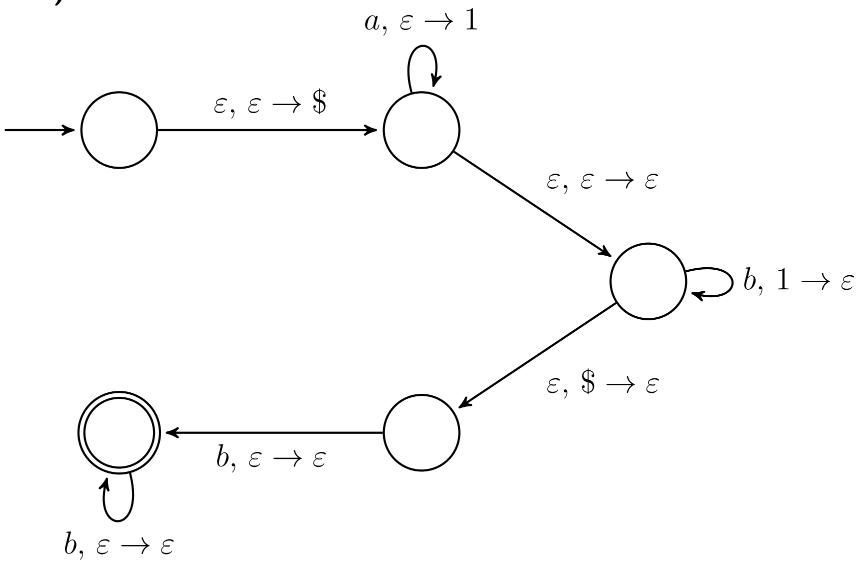
8–8) $L = \{a^k\, b^\ell\, c\, d^m\, e^n \mid \text{where} \quad k + \ell = m + n\}$

8–9) $L = \{a^n\, (bc)^{2m} a^{n+m} \mid n, m \geqslant 0\}$

8–10) $L = \{a^n\, b^m\, c^{2n+3m} \mid n, m \geqslant 0\}$

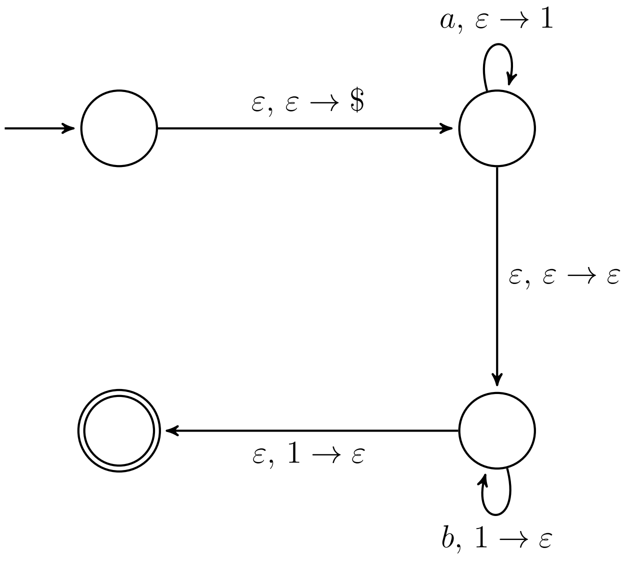
Find the language recognized by the following PDA:

8–11)



$a,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \varepsilon \to \varepsilon$
$b,\ 1 \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$

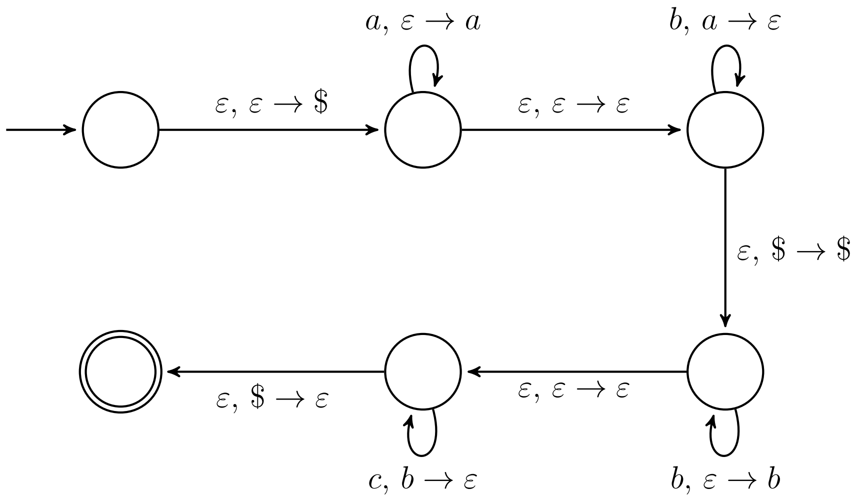
8–12)



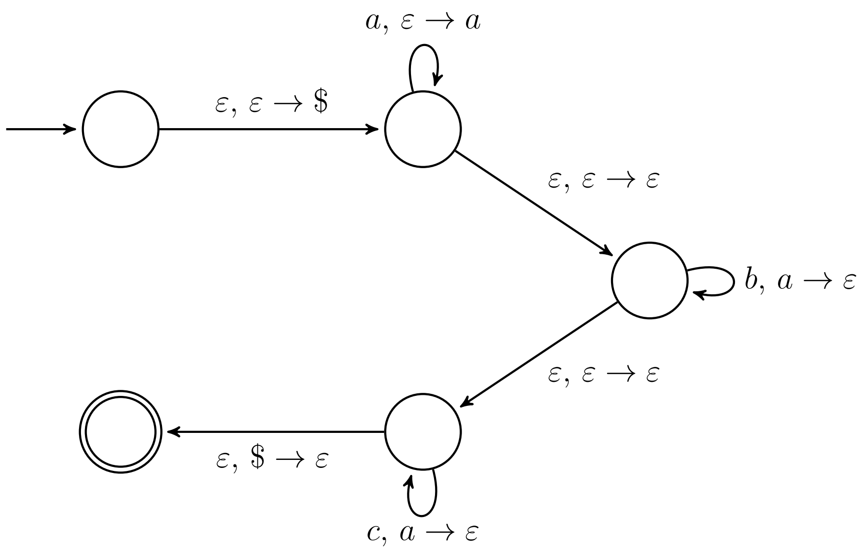
$a,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ 1 \to \varepsilon$
$b,\ 1 \to \varepsilon$

Find the language recognized by the following PDA:

8–13)

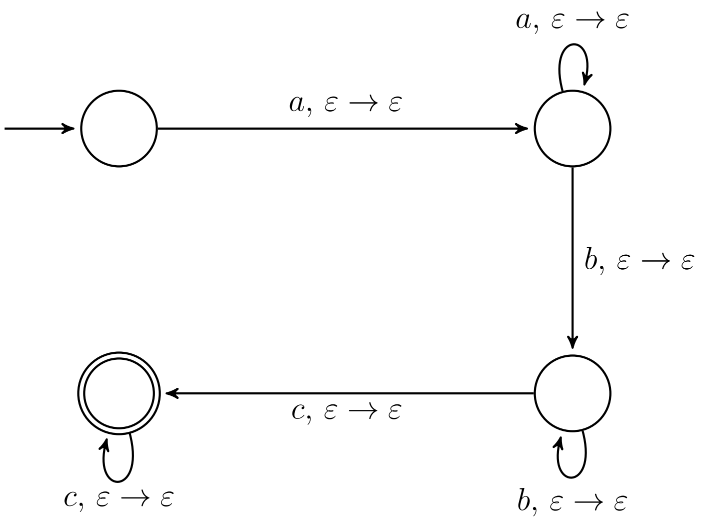

$a,\ \varepsilon \to a$
$b,\ a \to \varepsilon$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \$ \to \$$
$\varepsilon,\ \$ \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
$c,\ b \to \varepsilon$
$b,\ \varepsilon \to b$

8–14)

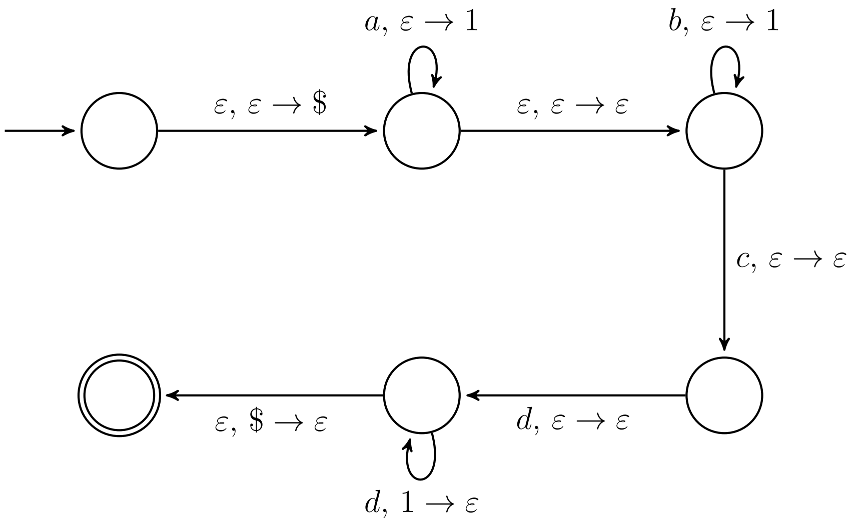

$a,\ \varepsilon \to a$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \varepsilon \to \varepsilon$
$b,\ a \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$
$c,\ a \to \varepsilon$

Find the languages recognized by the following PDAs:

8–15)

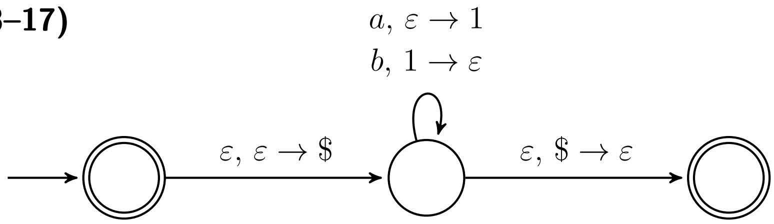

$a,\ \varepsilon \to \varepsilon$
$a,\ \varepsilon \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$
$c,\ \varepsilon \to \varepsilon$
$c,\ \varepsilon \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$

8–16)


$a,\ \varepsilon \to 1$
$b,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \varepsilon \to \varepsilon$
$c,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$
$d,\ \varepsilon \to \varepsilon$
$d,\ 1 \to \varepsilon$

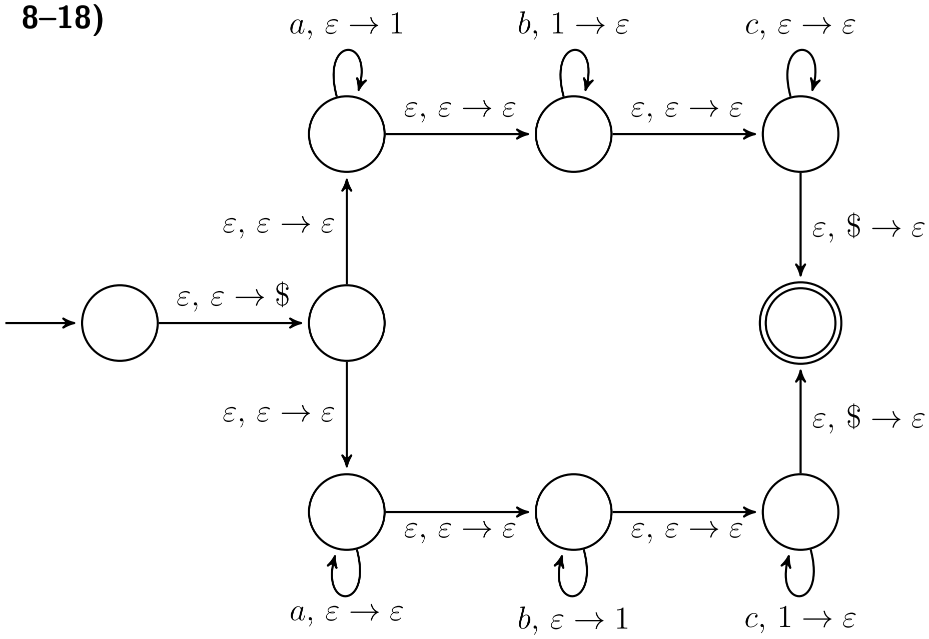
Find the languages recognized by the following PDAs:

8–17)



$a,\ \varepsilon \to 1$
$b,\ 1 \to \varepsilon$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \$ \to \varepsilon$

8–18)



$a,\ \varepsilon \to 1$
$b,\ 1 \to \varepsilon$
$c,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
$a,\ \varepsilon \to \varepsilon$
$b,\ \varepsilon \to 1$
$c,\ 1 \to \varepsilon$

8–19) Which of the following languages are context-free?

a) $L = \left\{ a^k\, b^\ell\, c^m\, d^n \;\middle|\; \text{where} \quad k + \ell + m \geqslant n \right\}$

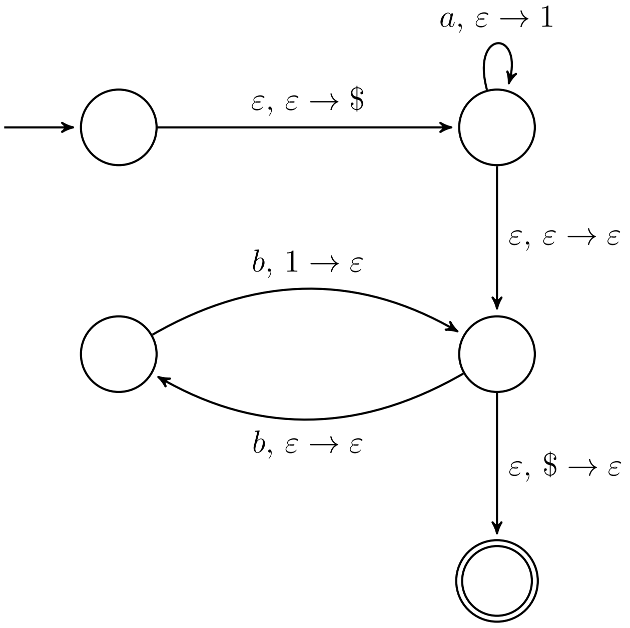
b) $L = \left\{ a^k\, b^\ell\, c^m\, d^n \;\middle|\; \text{where} \quad k < \ell < m < n \right\}$

c) $L = \left\{ a^k\, b^\ell\, c^m\, d^n \;\middle|\; \text{where} \quad k > \ell \quad \text{and} \quad m > n \right\}$

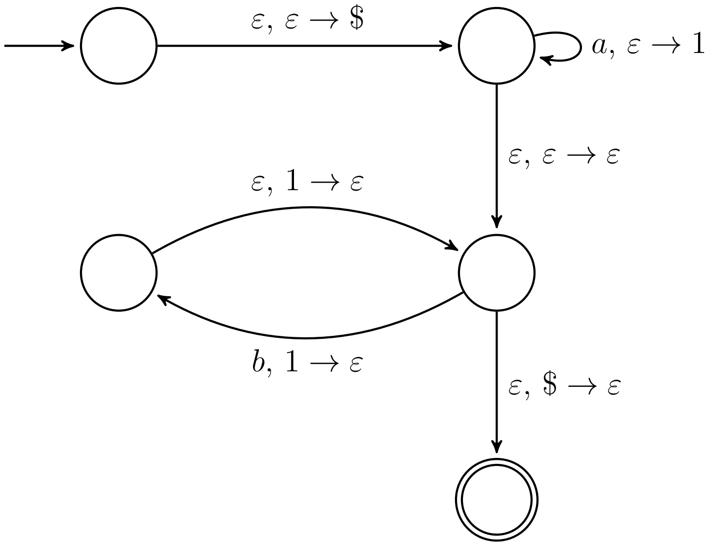
d) $L = \left\{ a^k\, b^\ell\, c^m\, d^n \;\middle|\; \text{where} \quad m = 2k \quad \text{and} \quad n = 3\ell \right\}$

ANSWERS

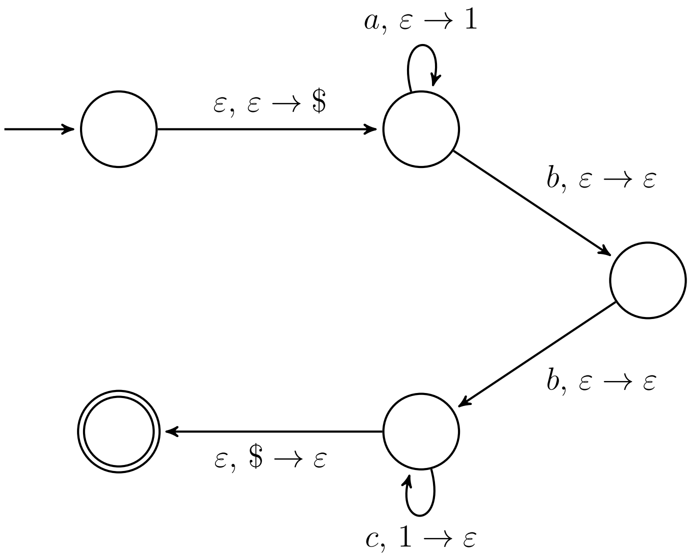
8–1)


$a,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \varepsilon \to \varepsilon$
$b,\ 1 \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$

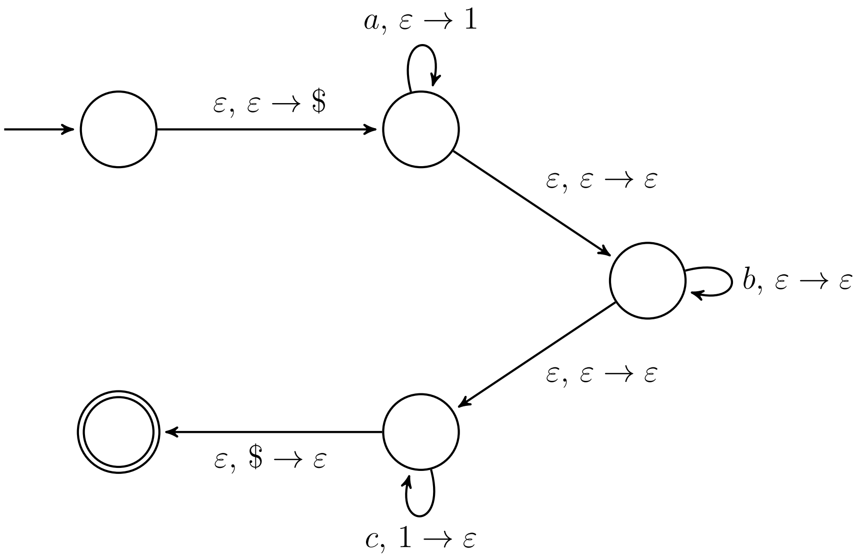
8–2)


$\varepsilon,\ \varepsilon \to \$$
$a,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ 1 \to \varepsilon$
$b,\ 1 \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$

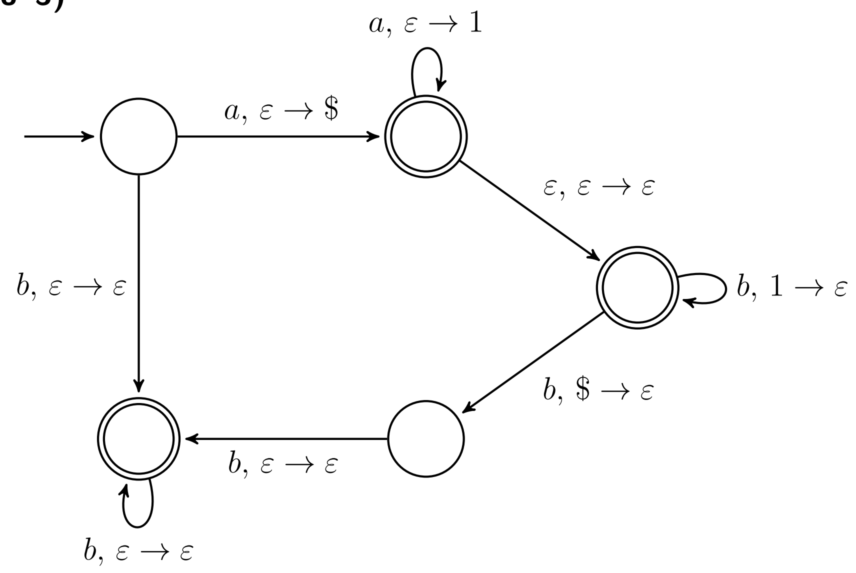
8–3)


$a,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to \$$
$b,\ \varepsilon \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$
$c,\ 1 \to \varepsilon$

8–4)

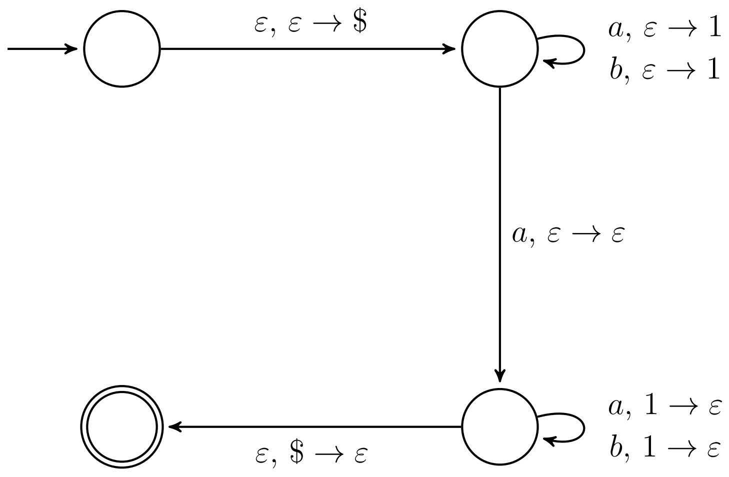

$a,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \varepsilon \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$
$c,\ 1 \to \varepsilon$

8–5)



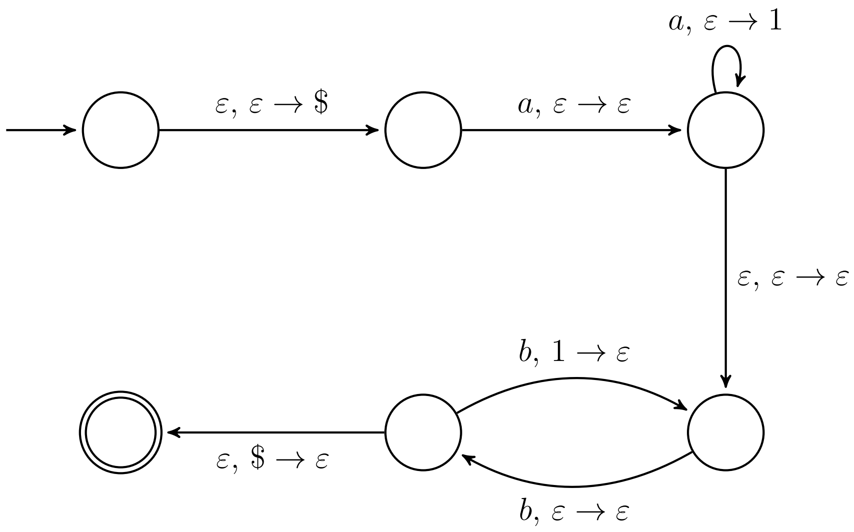
$a,\ \varepsilon \to 1$
$a,\ \varepsilon \to \$$
$\varepsilon,\ \varepsilon \to \varepsilon$
$b,\ 1 \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$
$b,\ \$ \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$

8–6)

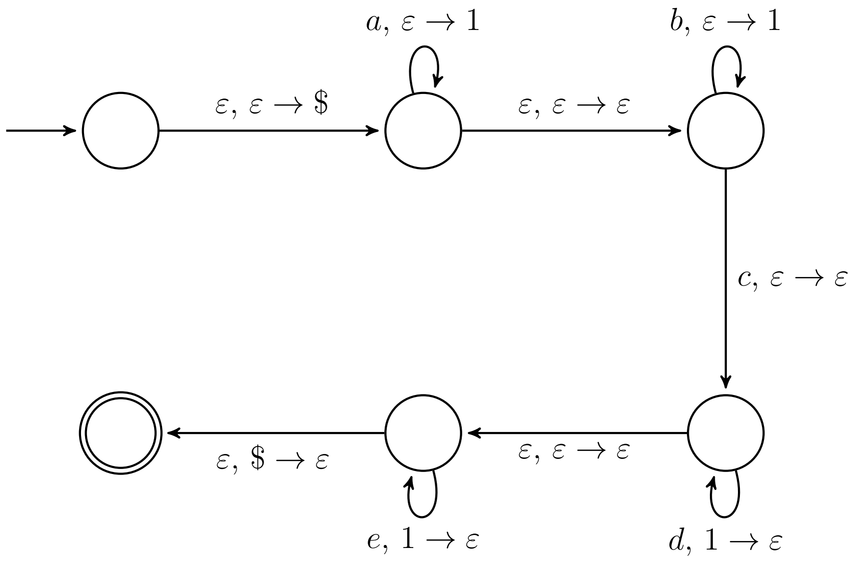


$\varepsilon,\ \varepsilon \to \$$
$a,\ \varepsilon \to 1$
$b,\ \varepsilon \to 1$
$a,\ \varepsilon \to \varepsilon$
$a,\ 1 \to \varepsilon$
$b,\ 1 \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$

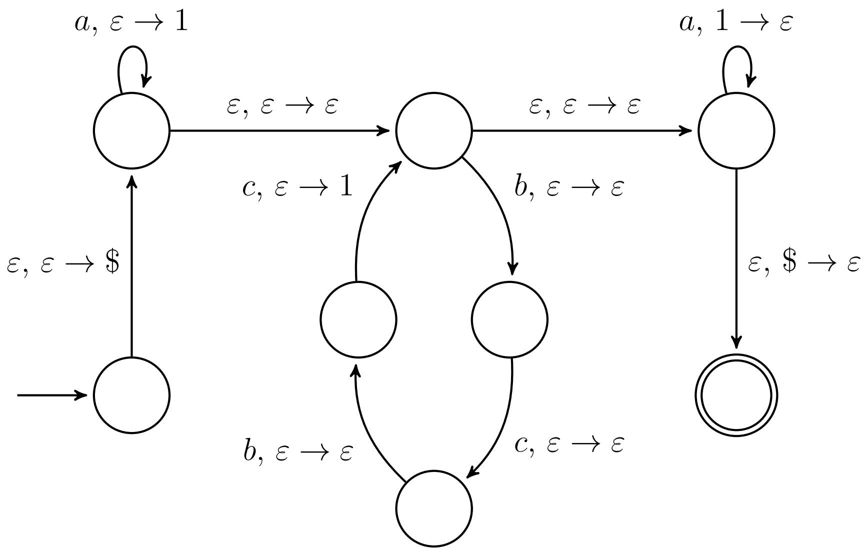
8–7)


$a,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to \$$
$a,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
$b,\ 1 \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$

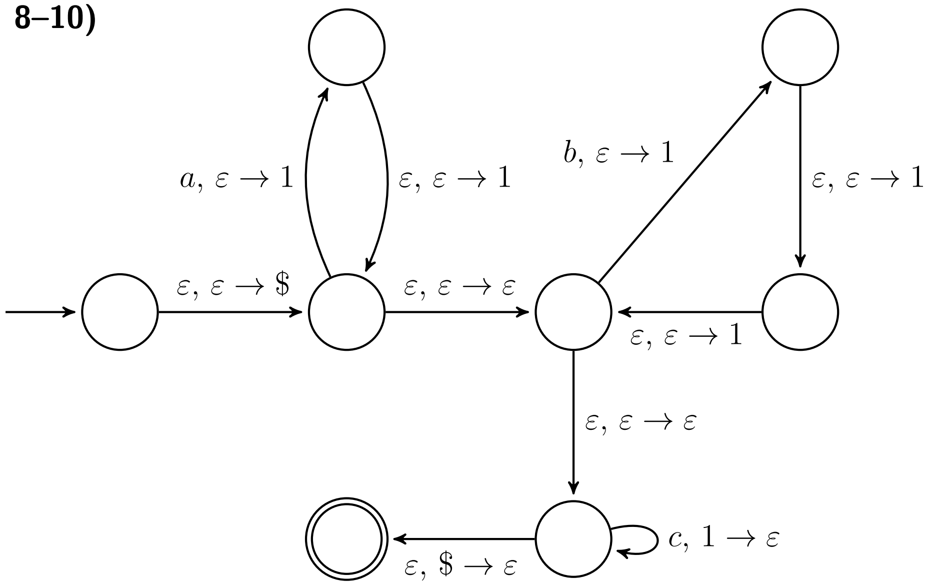
8–8)


$a,\ \varepsilon \to 1$
$b,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \varepsilon \to \varepsilon$
$c,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
$e,\ 1 \to \varepsilon$
$d,\ 1 \to \varepsilon$

8–9)


$a,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \varepsilon \to \varepsilon$
$a,\ 1 \to \varepsilon$
$c,\ \varepsilon \to 1$
$b,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \$ \to \varepsilon$
$b,\ \varepsilon \to \varepsilon$
$c,\ \varepsilon \to \varepsilon$

8–10)


$a,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to 1$
$b,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to \$$
$\varepsilon,\ \varepsilon \to \varepsilon$
$\varepsilon,\ \varepsilon \to 1$
$\varepsilon,\ \varepsilon \to \varepsilon$
$c,\ 1 \to \varepsilon$
$\varepsilon,\ \$ \to \varepsilon$

8–11) $L = \{a^n\, b^m \mid m > n\}$

8–12) $L = \{a^n\, b^m \mid n > m\}$

8–13) $L = \{a^m\, b^{m+n}\, c^n \mid m, n \geqslant 0\}$

8–14) $L = \{a^{m+n}\, b^m\, c^n \mid m, n \geqslant 0\}$

8–15) $L = \{a^k\, b^m\, c^n \mid k, m, n \geqslant 1\}$ Note that this PDA is actually a DFA. It does not use the stack at all.

8–16) $L = \{a^n\, b^m\, c\, d^{n+m+1} \mid m, n \geqslant 0\}$

8–17) The language consists of strings in $\{a, b\}^*$ where $n(a) = n(b)$. (Number of a's is equal to number of b's). Furthermore, at any point in the string counting from left to right $n(a) \geqslant n(b)$. This is like balanced parentheses.

8–18) $L = \{a^k\, b^m\, c^n \mid k, m, n \geqslant 0 \text{ and } (m = k \text{ or } m = n)\}$

8–19) Languages in parts a) and c) are context-free.

a) A CFG that generates the given language is:

$$\begin{aligned} S &\rightarrow aSd \mid AT \\ T &\rightarrow bTd \mid BU \\ U &\rightarrow cUd \mid C \\ A &\rightarrow aA \mid \varepsilon \\ B &\rightarrow bB \mid \varepsilon \\ C &\rightarrow cC \mid \varepsilon \end{aligned}$$

A PDA can be designed as follows: Push for a, b, c, pop for d. Accept if stack not empty when input ends.

c) A CFG that generates the given language is:

$$\begin{aligned} S &\rightarrow TU \\ T &\rightarrow aTb \mid A \\ U &\rightarrow cUd \mid C \\ A &\rightarrow aA \mid a \\ C &\rightarrow cC \mid c \end{aligned}$$

A PDA can be designed as follows: Push for a, pop for b. Continue if stack not empty. Reject if stack empty. Do the same for c and d.

Languages in parts b) and d) are not context-free. We can find neither a CFG nor a PDA for them. We will learn a method to prove this formally in the next chapter, but if you think about possible constructions and why they don't work, you will understand this topic better.

Week 9

Non-Context-Free Languages

9.1 PDA - CFG Equivalence

Remember the theorem in the previous chapter:

Theorem: A language is context-free if and only if there is a PDA recognizing it. In other words:

1. Given a CFG, we can find an equivalent PDA.
2. Given a PDA, we can find an equivalent CFG.

We will not give a full proof, which is quite complicated and can be found in more advanced textbooks. We will just illustrate the basic idea for the first part by an example:

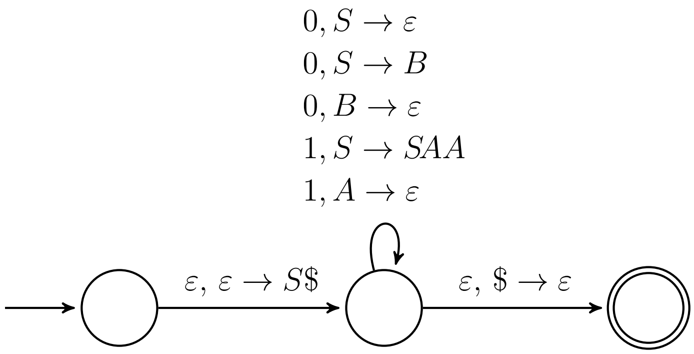
- Given CFG, find PDA.

First put the grammar in a special form where any production has the form $A \to aBCD \cdots$. That is, the right-hand side begins with a single terminal and continues with zero or more variables.

For example, given $S \rightarrow 1S11 \mid 0 \mid 00$

rewrite it as:
$$\begin{aligned} S &\rightarrow 1SAA \mid 0 \mid 0B \\ A &\rightarrow 1 \\ B &\rightarrow 0 \end{aligned}$$

The PDA must pop S and push SA for input 0, pop B for input 1 etc. Pushing two symbols to the stack can easily be accomplished by using an extra state.


$0, S \rightarrow \varepsilon$
$0, S \rightarrow B$
$0, B \rightarrow \varepsilon$
$1, S \rightarrow SAA$
$1, A \rightarrow \varepsilon$
$\varepsilon,\ \varepsilon \rightarrow S\$$
$\varepsilon,\ \$ \rightarrow \varepsilon$

(Push SA means first push A and then push S.)

This PDA simulates the leftmost derivation in the grammar. At each intermediate step, the string of variables is on the stack. Note that

- At each step, one digit from the input is read.
- If a variable is replaced by a terminal, that variable is popped from the stack. It disappears from derivation.
- If a variable is replaced by a string of variables, these variables are written to stack in correct order. (The leftmost is on top.)

The rules are chosen by the PDA nondeterministically and this corresponds to branching in the grammar.

9.2 Pumping Lemma for Context-Free Languages

Theorem: If L is a context-free language, then there is a number p called the pumping length where, if $s \in L$ and $|s| \geqslant p$, then it is possible to divide s into five pieces

$$s = uvxyz$$

satisfying the following conditions:

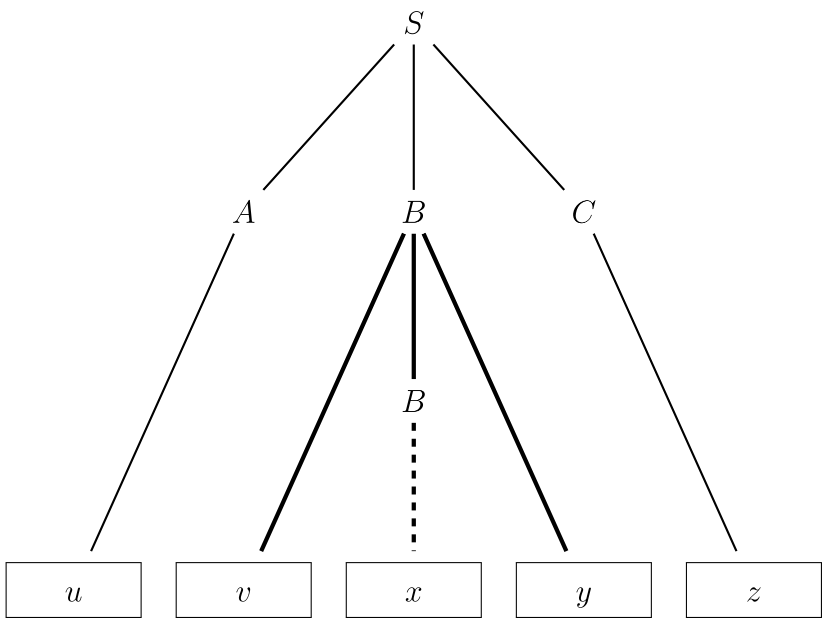
1. $uv^kxy^kz \in L$ for each $k \geqslant 0$.
2. $|vy| > 0$, and
3. $|vxy| \leqslant p$.

Condition 2 states that v and y cannot both be ε. Otherwise, the theorem would be true but hollow.
Condition 3 states that the pieces v, x, and y (when concatenated) have length at most p.
For example, if L is the language of palindromes of odd length, then $p = 3$. Any string of length 3 or more can be pumped.

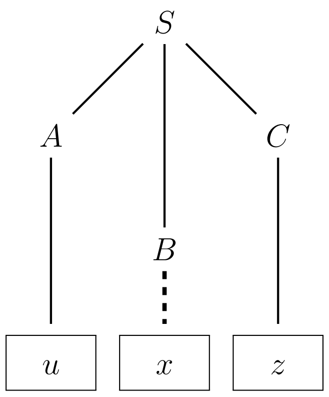
	u	v	x	y	z
$s = 101$:	ε	1	0	1	ε
$s = 11011$:	1	1	0	1	1
$s = 011010110$:	011	0	1	0	110

Similarly, in the language $\{a^nb^n \mid n \geqslant 0\}$ we can partition any string of length 2 or greater into those five parts easily.

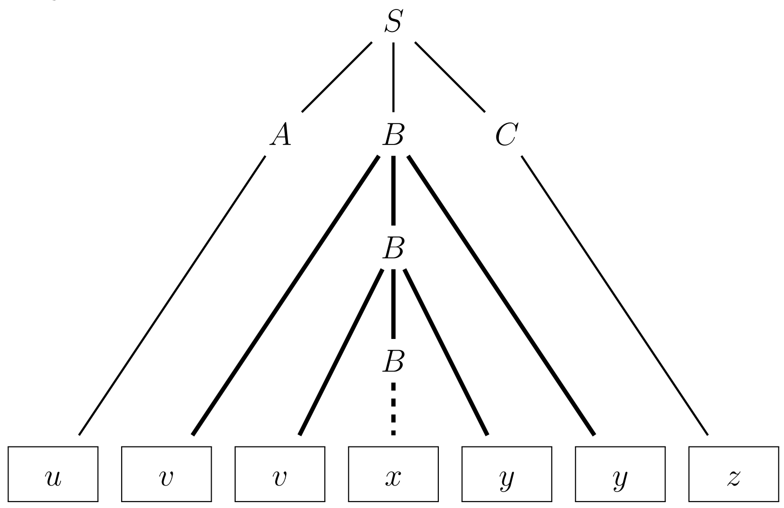
Sketch of Proof: Let L be a CFL and let $s \in L$ be a sufficiently long string. Then, some variable symbol B must repeat in the parse tree because of the pigeonhole principle.



Delete the first B:



Or copy and paste B:



We can continue to copy and paste. In both cases, the resulting string must be part of the language because it is derived using the grammar.

Example 9–1: Use pumping lemma to show that L is not a CFL:

$$L = \left\{0^n 10^n 10^n \;\middle|\; n \geqslant 0\right\}$$

Solution: Assume L is context-free. Let the pumping length be p. Consider the string $s = 0^p 10^p 10^p$. If we partition it as

$$s = 0^p 10^p 10^p = uvxyz$$

then v and y cannot contain 1. Otherwise, uv^2xy^2z will contain more than two 1's so it won't be in L. Therefore they must consist of 0's. But, once again, $uv^2xy^2z \notin L$ because powers of 0's will be different.

So s cannot be pumped and L is not a CFL.

Example 9–2: Using the pumping lemma, show that the following language is not context-free.

$$L = \left\{ a^n b^n c^n \;\middle|\; n \geqslant 0 \right\}$$

Solution: Suppose L is context-free. Let p be the pumping length. Choose test string s as $s = a^p b^p c^p$.
Now we have to partition s such that

$$a^p b^p c^p = uvxyz$$

How can we do that? In other words, what are v and y?

1. They contain more than one symbol.

   (For example, v may contain some a's and some b's.) In this case, the pumped string uv^2xy^2z will have symbols out of order. (For example, there will be a's after b's.) Therefore

   $$uv^2xy^2z \notin L$$

2. They contain a single symbol.

   In that case, we can pump at most two of the three symbols $\{a, b, c\}$ using v and y. The remaining symbol will still have power n in the pumped string. For example, if we choose $v = a$ and $y = b$ we obtain

   $$uv^2xy^2z = a^{n+1} b^{n+1} c^n \notin L$$

Therefore we cannot pump this string. By the pumping lemma, L is not context-free.

Example 9–3: Using the pumping lemma, show that the following language is not context-free.

$$L = \left\{a^k b^m c^n \;\middle|\; k < m < n\right\}$$

Solution: Suppose L is context-free. Let the pumping length be p. Consider the test string

$$s = a^p b^{p+1} c^{p+2}$$

Let $s = uvxyz$. If v or y contain more than one type of symbol, $uv^2xy^2z \notin L$ because symbols are out of order. Therefore v and y must contain a single type of symbol only. (Assume neither is ε for parts 1 and 2.)

1. v contains a's. Pump up once or if necessary twice. If y contains b's, we will have more a's than c's. If y contains c's, we will have more a's than b's. If y contains only a's both of these are correct. Therefore the resulting string is not in L.

2. v consist of b's or c's. We have to be careful here. It is possible to choose y such that it consists of c's and then as we pump up, the string is in L. But in this case, we cannot pump down, because the inequality condition between a's and b's (or a's and c's, or b's and c's) will be broken after pumping once.

3. $v = \varepsilon$ or $y = \varepsilon$. Similar to previous cases, left as an exercise.

The string s cannot be pumped so L is not context-free.

Example 9–4: Are the following languages context-free?

a) $A = \{a^n\, b^m\, c^m\, d^n \mid n, m \geqslant 0\}$

b) $B = \{a^n\, b^n\, c^m\, d^m \mid n, m \geqslant 0\}$

c) $C = \{a^n\, b^m\, c^n\, d^m \mid n, m \geqslant 0\}$

If they are, find a CFG that generates them. If not, prove it using the pumping lemma.

Solution: a) We can easily find a CFG for the language A:

$$\begin{aligned} S &\rightarrow aSd \mid T \\ T &\rightarrow bTc \mid \varepsilon \end{aligned}$$

Therefore it is a CFL. But let's look at that language in different ways. Can we find a PDA for it?

Yes we can! It will push a's into stack for each a, then push b's for each b, then pop b's for each c and pop a's for each d.

If we use the pumping lemma, we see that it is satisfied for all sufficiently long strings in this language. (But this is NOT a proof that A is a CFL.)
If $m \geqslant 1$, choose $v = b$, $x = \varepsilon$, $y = c$.
If $m = 0, n \geqslant 1$, choose $v = a$, $x = \varepsilon$, $y = d$.
If $m = n = 0$, the string is not sufficiently long and cannot be pumped.

b) Similarly, B is a CFL:

$$\begin{aligned} S &\rightarrow TU \\ T &\rightarrow aTb \mid \varepsilon \\ U &\rightarrow cUd \mid \varepsilon \end{aligned}$$

A PDA for B will work as follows:

Push a's into stack for each a, then pop a's for each b. Push c's for each c, then pop c's for each d.

Any long string in B can be pumped as follows:
If $n \geqslant 1,$ choose $v = a,\ x = \varepsilon,\ y = b.$
If $n = 0, m \geqslant 1,$ choose $v = c,\ x = \varepsilon,\ y = d.$
If $m = n = 0,$ the string is not sufficiently long.

c) It is not possible to find a CFG or PDA for the language C. Our inability to find these cannot be considered as proof that they do not exist, but if you think about this problem, you will understand CFGs and PDAs better.

Let's use the pumping lemma for a definitive conclusion. Suppose C is a CFL. Let p be the pumping length. Choose test string as

$$s = a^n\, b^m\, c^n\, d^m$$

where $m, n > p$. Now divide it into five parts:

$$a^n\, b^m\, c^n\, d^m = uvxyz$$

If v or y contain more than one type of symbol, $uv^2xy^2z \notin C$ because symbols are out of order. Therefore v and y must contain a single type of symbol only.

There are two possibilities:

- v consists of a's and y consists of c's, or
- v consists of b's and y consists of d's.

Both cases violate the rule $|vxy| \leqslant p$.

Example 9–5: Using the pumping lemma, show that the following language is not context-free.

$$L = \{ww \mid w \in \{0,1\}^*\}$$

Solution: This language consists of strings of even length that repeat itself. For example,

$$10011001$$

or

$$1000010000$$

(Is it possible to find a CFG or PDA for this language? Please think about it before proceeding.)

Suppose L is context-free. Let p be the pumping length. Choose test string s as

$$s = 0^p10^p1$$

Divide it as

$$0^p10^p1 = uvxyz$$

If we choose $v = 0$ and $y = 0$, then, this string can always be pumped. In other words, the string

$$uv^kxy^kz = 0^{p+k-1}\,1\,0^{p+k-1}\,1$$

will be in L for all k including $k = 0$.

So the string s above was a bad choice, as we could not reach any conclusion.

Note that operations above do NOT prove L is context-free. They don't prove anything!

Let's try again and choose a new test string:

$$s = 0^p 1^p 0^p 1^p$$

Partitioning gives:

$$0^p 1^p 0^p 1^p = uvxyz$$

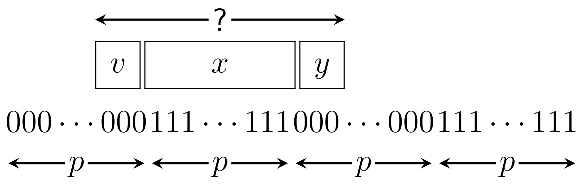
If v and y were different, for example, $v = 00, y = 01$, then, the string obtained after pumping would not be of the form ww. Therefore, they have to be identical.

For example, if $v = y = 0$ or $v = y = 111$ it is possible to pump this string and keep the form ww.

But considering the third condition of the pumping lemma

$$|vxy| \leqslant p$$

we see that it is impossible to choose v and y in that way.



In other words, if v consists of 0's, then, y must consist of 1's. But in this case,

$$uv^2xy^2z \notin L$$

We cannot pump this string therefore L is not context-free.

Remark: Do not forget that the pumping lemma says:

$$L \text{ is context-free} \quad \Rightarrow \quad L \text{ is pumpable}$$

This is logically equivalent to:

$$L \text{ is NOT pumpable} \quad \Rightarrow \quad L \text{ is NOT context-free}$$

But it is not equivalent to

$$L \text{ is pumpable} \quad \Rightarrow \quad L \text{ is context-free}$$

This last statement is wrong. There are some non-context-free languages that are pumpable.

Example 9–6: Are the following languages context-free? What can we say using the pumping lemma?

a) $A = \left\{0^n 1^m 0^{3n} \;\middle|\; n, m \geqslant 0\right\}$

b) $B = \left\{0^n 1 0^m 1 0^{n+m} \;\middle|\; n, m \geqslant 0\right\}$

Solution: We can easily pump any long string in these languages.

- For A, pumping length is 4. Given any string $s \in A$, $|s| \geqslant 4$:
  If $m \geqslant 1$ choose $v = 1$ and $y = \varepsilon$.

  $$uv^k xy^k z = 0^n 1^k 1^{m-1} 0^{3n} \in A$$

  ($x = \varepsilon$, $v = y = 1$ would also work for $m \geqslant 2$.)

  If $m = 0$ then $n \geqslant 1$ choose $v = 0$, $y = 000$.

  $$uv^k xy^k z = 0^{n-1} 0^k 0^{3k} 0^{3n-3} \in A$$

- For B, pumping length is again 4. Choose $v = 0$ (left or middle) and $y = 0$ (right).

For example, $s = 0^3 1 0^2 1 0^5$

$$\begin{array}{ccccc} u & v & x & y & z \\ \boxed{00010} & \boxed{0} & \boxed{1} & \boxed{0} & \boxed{0000} \end{array}$$

$s = 0^4 1 1 0^4$

$$\begin{array}{ccccc} u & v & x & y & z \\ \boxed{000} & \boxed{0} & \boxed{11} & \boxed{0} & \boxed{000} \end{array}$$

What we said about A and B is correct but it is not a proof that these languages are context-free.

The pumping lemma does not say this. It is a mistaken assumption to think that if L satisfies conditions of the pumping lemma, then it is CFL. Pumping lemma says *if L is CFL, then it satisfies those conditions*.

As usual, the correct way to answer such a question is by finding a CFG. We can easily see that the grammars are:

$$\text{Language} \quad A: \qquad \begin{array}{lcl} S & \rightarrow & 0S000 \mid T \\ T & \rightarrow & 1T \mid \varepsilon \end{array}$$

$$\text{Language} \quad B: \qquad \begin{array}{lcl} S & \rightarrow & 0S0 \mid 1T \\ T & \rightarrow & 0T0 \mid 1 \end{array}$$

Can you see the relationship between these productions and the way we chose v and y when pumping?

Example 9–7: Using the pumping lemma, show that the following languages are not context-free.

a) $A = \{a^{n!} \mid n \geqslant 0\}$

b) $B = \{a^{n^2} \mid n \geqslant 0\}$

Solution: a) Assume A is context-free. Let the pumping length be p. Choose the test string as $s = a^{p!}$. Divide it into five parts as

$$a^{p!} = uvxyz$$

Using the pumping lemma, Condition 3:

$$|vxy| \leqslant p \quad \Rightarrow \quad |uv^2xy^2z| \leqslant p! + p$$

We know that: $p! + p < (p+1)!$ (Why?)

$$\Rightarrow \quad uv^2xy^2z \neq a^{(p+1)!} \quad \Rightarrow \quad uv^2xy^2z \notin L$$

Therefore s cannot be pumped so A is not context-free.

b) Assume B is context-free. Let the pumping length be p. Choose the test string as $s = a^{p^2}$. Divide it into five parts as

$$a^{p^2} = uvxyz$$

Using the pumping lemma, Condition 3:

$$|vxy| \leqslant p \quad \Rightarrow \quad |uv^2xy^2z| \leqslant p^2 + p$$

We know that: $p^2 + p < (p+1)^2$ (Why?)

$$\Rightarrow \quad uv^2xy^2z \neq a^{(p+1)^2} \quad \Rightarrow \quad uv^2xy^2z \notin L$$

Therefore s cannot be pumped so B is not context-free.

9.3 Closure Properties of CFLs

We know that the set of regular languages is closed under the operations of union, concatenation, and star. We can say the same thing for context-free languages:

Theorem: If A and B are CFLs, then

1. $A \cup B$
2. $A \circ B$
3. A^*

are also CFLs.

Sketch of Proof: If A and B are context-free, then each must have a CFG. Suppose these are given as:

$$\begin{array}{rrcl} A: & S_1 & \to & X_1X_2 \\ & X_1 & \to & X_3X_4 \\ & & \vdots & \end{array} \qquad \begin{array}{rrcl} B: & S_2 & \to & X_5X_6 \\ & X_5 & \to & X_7X_8 \\ & & \vdots & \end{array}$$

Then, the following constructions will generate the composite languages:

$$\begin{array}{rrcl} A \cup B: & S_0 & \to & S_1 \mid S_2 \\ & S_1 & \to & X_1X_2 \\ & X_1 & \to & X_3X_4 \\ & & \vdots & \\ & S_2 & \to & X_5X_6 \\ & X_5 & \to & X_7X_8 \\ & & \vdots & \end{array} \qquad \begin{array}{rrcl} A \circ B: & S_0 & \to & S_1S_2 \\ & S_1 & \to & X_1X_2 \\ & X_1 & \to & X_3X_4 \\ & & \vdots & \\ & S_2 & \to & X_5X_6 \\ & X_5 & \to & X_7X_8 \\ & & \vdots & \end{array}$$

$$\begin{aligned} A^*: \quad S_0 &\to S_0 S_1 \mid \varepsilon \\ S_1 &\to X_1 X_2 \\ X_1 &\to X_3 X_4 \\ &\vdots \end{aligned}$$

Note that the sets of variables for $A \cup B$ and $A \circ B$ must be disjoint. If necessary, we may rename some of them.

The set of regular languages is also closed under intersection and complement. But we cannot say the same thing for CFLs. For example,

$$A = \{a^n\, b^n\, c^m \mid n, m \geqslant 0\}$$

and

$$B = \{a^n\, b^m\, c^m \mid n, m \geqslant 0\}$$

are context-free. But their intersection

$$A \cap B = \{a^n\, b^n\, c^n \mid n \geqslant 0\}$$

is not a CFL. (We can prove this by the pumping lemma.)

Suppose the set of CFLs is closed under complementation. Then, if A and B are CFLs:

$$\begin{array}{rcl} \overline{A} \text{ and } \overline{B} & \text{are} & \text{also CFLs} \\ \overline{A} \cup \overline{B} & \text{is} & \text{also CFL} \\ \overline{\overline{A} \cup \overline{B}} & \text{is} & \text{also CFL} \end{array}$$

But we know that

$$\overline{\overline{A} \cup \overline{B}} = A \cap B$$

so we have a contradiction. Therefore the complement of a CFL is not necessarily a CFL.

EXERCISES

Are the following languages context-free?
If your answer is *Yes*, give a CFG generating the language.
If your answer is *No*, prove your claim using the pumping lemma.

9–1) $L = \{a^{2n}\, b^{n+1}\, c^{n-1} \mid n \geqslant 1\}$

9–2) $L = \{a^n\, b^n\, c^{2n}\, d^{2n} \mid n \geqslant 0\}$

9–3) $L = \{a^n\, b^m\, c^{2n}\, d^{2m} \mid m, n \geqslant 0\}$

9–4) $L = \{a^n\, b^m\, c^{n-m}\, d^3 \mid n > m \geqslant 0\}$

9–5) $L = \{0^n\, 1^m \mid m > n + 4\}$

9–6) $L = \{0^n 1 0^{4n} 1 0^{2n} \mid n \geqslant 0\}$

9–7) $L = \{0^n\, 1^m \mid m = n^2\}$

9–8) $L = \{0^n\, 1^m \mid n \neq m\}$

Are the following languages context-free?
If your answer is *Yes*, give a CFG generating the language.
If your answer is *No*, prove your claim using the pumping lemma.

9–9) $L = \{a^n\, b^m\, c^n \mid n \text{ is odd, } m \text{ is even}\}$

9–10) $L = \{a^k\, b^m\, c^n \mid n > m \text{ and } n > k\}$

9–11) $L = \{a^k\, b^m\, c^n \mid k > m > n > 0\}$

9–12) $L = \{a^k\, b^m\, c^n \mid n = k + 2m + 1\}$

9–13) $L = \{a^k\, b^m\, c^n \mid k = m \text{ and } k \neq n\}$

9–14) $L = \{a^n\, b^m \mid n + m = 4\}$

9–15) $L = \{a^k\, b^m\, c^m\, b^n\, c^n\, a^k \mid k, m, n \geqslant 0\}$

9–16) $L = \{a^n\, b^m\, c^{nm} \mid m, n \geqslant 1\}$

ANSWERS

9–1) Not context-free.

9–2) Not context-free.

9–3) Not context-free.

9–4) Context-free.

$$\begin{array}{rcl} S & \rightarrow & Addd \\ A & \rightarrow & aAc \mid aBc \\ B & \rightarrow & aBb \mid \varepsilon \end{array}$$

9–5) Context-free.

$$\begin{array}{rcl} S & \rightarrow & 0S1 \mid T \\ T & \rightarrow & 1T \mid 11111 \end{array}$$

9–6) Not context-free.

9–7) Not context-free.

9–8) Context-free.

$$\begin{array}{rcl} S & \rightarrow & 0S1 \mid A \mid B \\ A & \rightarrow & 0A \mid 0 \\ B & \rightarrow & B1 \mid 1 \end{array}$$

9–9) Context-free.

$$\begin{aligned} S &\rightarrow aaScc \mid aTc \\ T &\rightarrow bTb \mid \varepsilon \end{aligned}$$

9–10) Not context-free.

9–11) Not context-free.

9–12) Context-free.

$$\begin{aligned} S &\rightarrow aSc \mid T \\ T &\rightarrow bTcc \mid c \end{aligned}$$

9–13) Not context-free. (Hint: $s = a^p\, b^p\, c^{p+p!}$)

9–14) Context-free.

$$S \rightarrow aaaa \mid aaab \mid aabb \mid abbb \mid bbbb$$

9–15) Context-free.

$$\begin{aligned} S &\rightarrow aSa \mid AB \\ A &\rightarrow bAc \mid \varepsilon \\ B &\rightarrow bBc \mid \varepsilon \end{aligned}$$

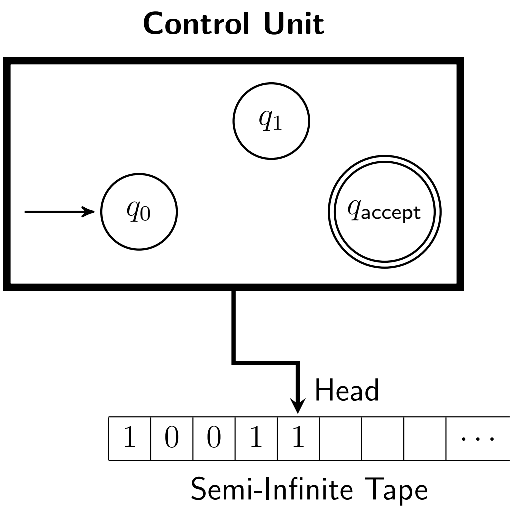
9–16) Not context-free. (Hint: $s = a^p\, b^p\, c^{p^2}$)

Week 10

Turing Machines - I

10.1 Definition and Diagrams

A **Turing Machine (TM)** is like a finite automaton but it has unlimited memory, and it can access any part of the memory. The head can move left or right. Thus, it can read or write on any part of the tape. This is much better than a PDA!



A Turing machine (TM) receives its input on the leftmost part of the tape. The rest of the tape is initially blank. The head cannot move to the left of the start square.

Like a DFA, the computation proceeds deterministically according to the transition function. In the beginning, the head is on the leftmost cell. At each step, it can read and write on the tape, then moves left or right and the controller may change state.

Formal Definition of a Turing Machine:
A Turing machine is an 8-tuple, $(Q, \Sigma, \Gamma, \delta, \Box, q_0, q_{\text{accept}}, q_{\text{reject}})$ where Q, Σ, Γ are all finite sets and:

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the tape alphabet, where $\Sigma \subseteq \Gamma \setminus \Box$,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $\Box$ is the blank symbol. (Note that $\Box \notin \Sigma$ and $\Box \in \Gamma$.)
6. $q_0 \in Q$ is the start state,
7. $q_{\text{accept}} \in Q$ is the accept state, and
8. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

If the machine enters an accept or reject state, the computation halts. (It stops immediately.) There is the danger that the machine loops and the computation never halts. We will analyze this situation later. A TM may work as a DFA and just accept or reject given strings. But in addition, it can also give an output on the tape like a computer.

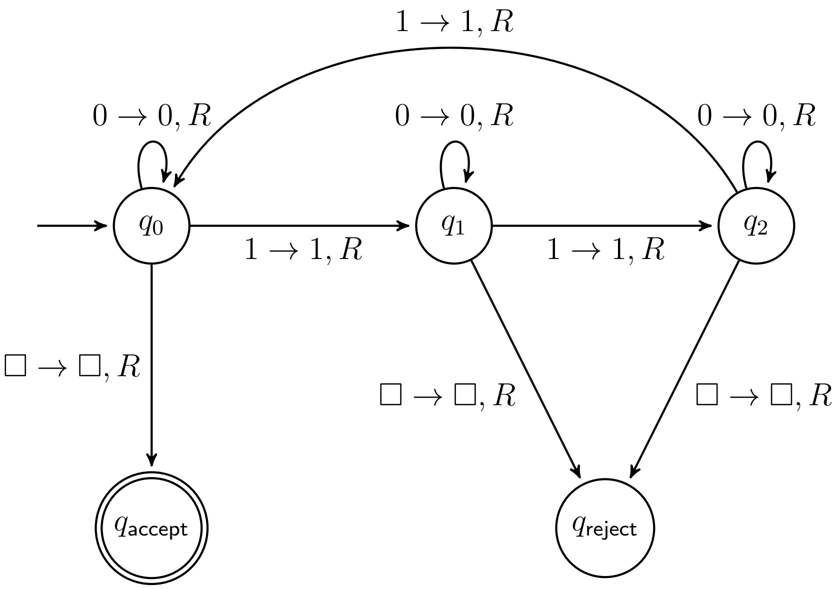
Example 10–1: Design a TM that recognizes the language

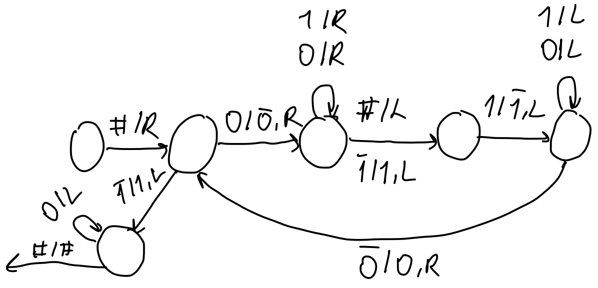
$$L = \{ w \mid \text{Number of } 1\text{'s in } w \text{ is a multiple of } 3 \},$$

over the input alphabet $\Sigma = \{0, 1\}$.

Solution: This is very much like a DFA. The machine will only read. It will not write anything on tape. There will be three states for counting modulo 3.

- At each 1, it will change state.
- At 0's, nothing will change.
- At $\square$, the machine meets the end of input.





#0011#

$\bar{0}$ $\bar{1}$

0 $\bar{0}$ $\bar{1}$ 1

0

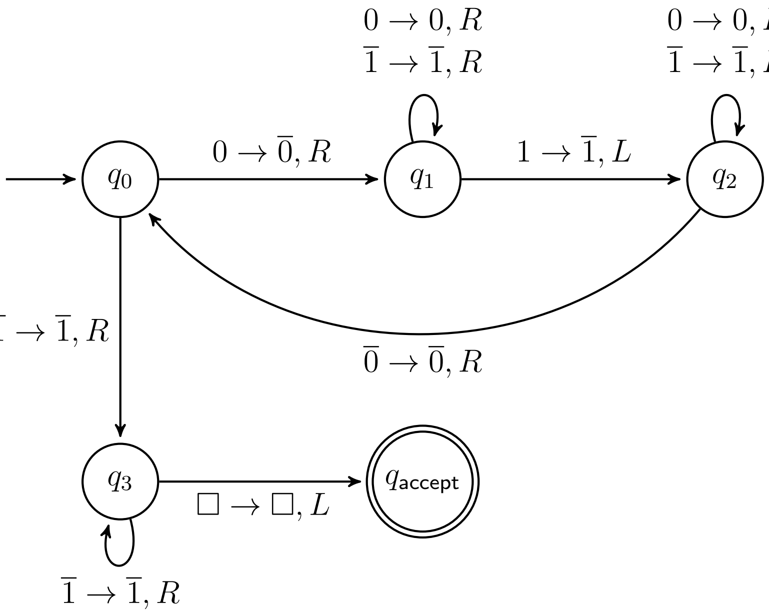
#000111#

Example 10–2: Design a TM that recognizes the language

$$L = \{0^n 1^n \mid n \geqslant 1\}$$

Solution: In this example, we have to write on tape. Tape and input alphabets will be different. In addition to 0 and 1, we will use symbols $\overline{0}$ and $\overline{1}$. (Marked zero and one.)

The idea is to cross 0's and 1's one by one: Mark a 0, move right until first unmarked 1, mark it, move left until a marked 0, loop.



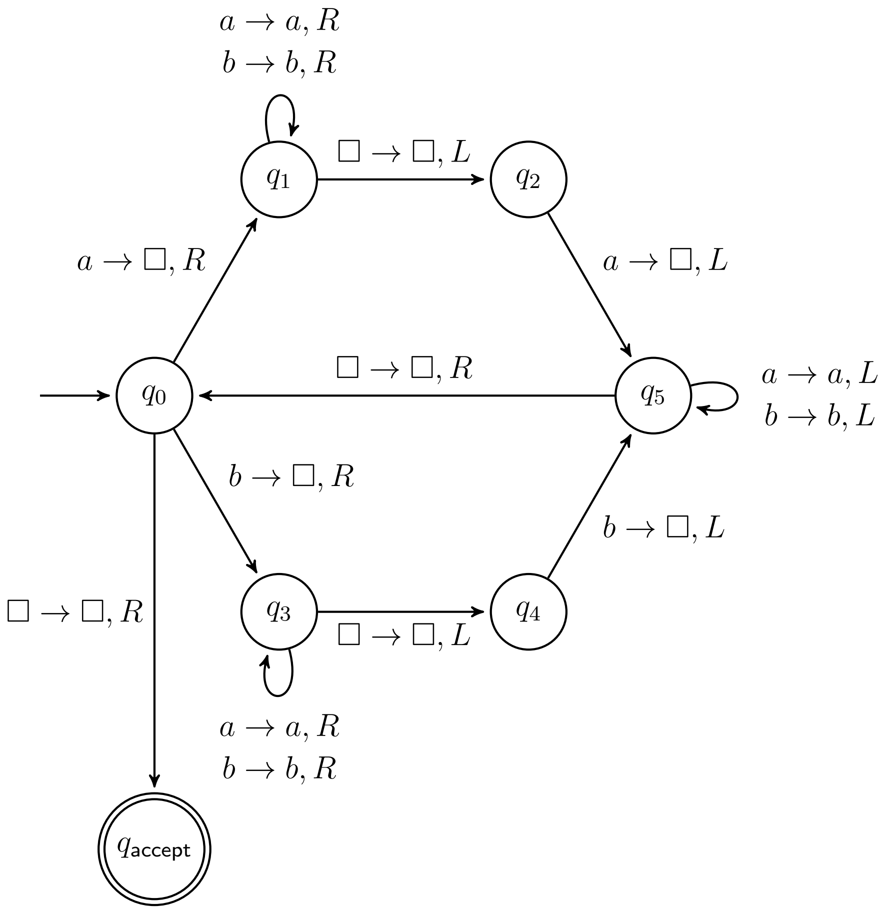
$0 \to 0, R$
$\overline{1} \to \overline{1}, R$
$0 \to 0, L$
$\overline{1} \to \overline{1}, L$
$0 \to \overline{0}, R$
$1 \to \overline{1}, L$
q_0
q_1
q_2
$\overline{1} \to \overline{1}, R$
$\overline{0} \to \overline{0}, R$
q_3
q_{accept}
$\square \to \square, L$
$\overline{1} \to \overline{1}, R$

The reject state is not shown, but like an NFA, the machine moves to the reject state (in other words, it crashes) if it receives an unexpected input, for example, $\square$ at q_1.

Example 10–3: Design a TM that recognizes the language of palindromes of even length over the alphabet $\Sigma = \{a, b\}$.

Solution: The basic idea is to delete a symbol at the start, go to the end, delete the same symbol at the end, return to (new) start cell. If you find a different symbol at the end, crash.

Also crash if you find nothing at the end, after deleting the first symbol. This means odd length.



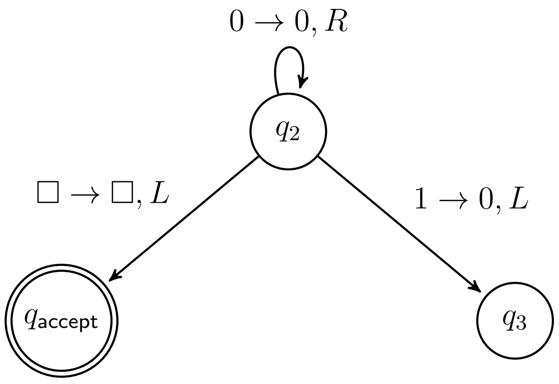
10.2 Algorithmic Description of TMs

It is also possible to use a TM as a **transducer** rather than a language recognizer. A transducer produces a desired output for a given input.

Turing Machines are models of computation that can do whatever a computer can do. That's why they were developed. TM is a very effective theoretical tool.

But, TM diagrams are more complicated than DFA or PDA diagrams. **Therefore we will use a higher level pseudo-code description from now on**. In other words, we will think of a TM as an algorithm.

For example, the following fragment of a TM



will be expressed as:

```
Move right over $0$'s.
If you read $1$
    Replace $1$ by $0$,
    Move head one unit left.
If you read $\square$
    ACCEPT
```

Example 10–4: Give a description of a Turing machine M that, given an input from $\{1\}^*$, doubles the number of symbols.

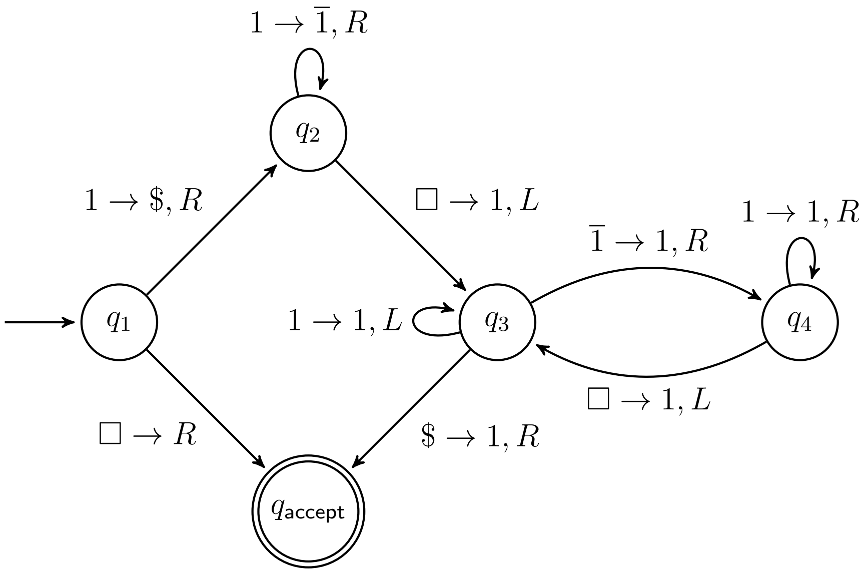
Solution:

1. If the first symbol is $\square$, ACCEPT.
   If it is 1, replace by \$ and move right.

2. Put a mark on all 1's until meeting $\square$.
   Write 1.

3. Move left over 1's until meeting the first $\overline{1}$.
   Unmark it. $(\overline{1} \to 1)$.
   If you meet \$, replace by 1, ACCEPT.

4. Move right over 1's until meeting $\square$. Write 1.
   Go To 3.



This is a transducer!

Example 10–5: Describe (in algorithm format) a TM that recognizes the language

$$L = \left\{ a^{2n} \, b^n \;\middle|\; n \geqslant 0 \right\}$$

Solution:

```
1. Sweep from start to end.
   If there is any $a$ after first $b$
         REJECT.

2. Move head to start. Search for $a$.
   If found
         Cross it. (Replace by $\times$)
         Search for $a$.
         If found
              Cross it.
         Else
              REJECT.
   Else
         Go To $4$.

3. Search for $b$.
   If found
         Cross it.
         Go To $2$.
   Else
         REJECT.

4. Move head to start. Search for $b$.
   If found
         REJECT.
   Else
         ACCEPT.
```

Example 10–6: Describe a TM recognizing $L = \left\{ a^n \, b^{n^2} \;\middle|\; n \geqslant 1 \right\}$.

Solution:

```
1. Sweep from left to right. If there is no $a$, or no $b$,
   or if they are out of order,   REJECT.

2. Move head to start. Search for $a$.
   If found,
         Mark it.      //$(a \to \overline{a})$
         Move head to end. Write $c$ to tape.
         Go To $2$.
   //Write as many $c$'s as there are $a$'s.

3. Move head to start. Search for $c$.
   If found
         Delete it,    Go To $4$.
   Else
         Move head to start. Search for $b$.
         If found,   REJECT.
         Else,   ACCEPT.

4. Unmark all marked $a$'s.        //$(\overline{a} \to a)$

5. Move head to start. Search for $a$.
   If found
         Mark it.
         Search for $b$.
         If found
               Mark it,    Go To $5$.
         Else
               REJECT.
   Else
         Go To $3$.
```

Example 10–7: Describe a TM that recognizes the language

$$L = \left\{ wcw \;\middle|\; w \in \{a,b\}^* \right\}$$

This language consists of two identical strings of a's and b's separated by single c. For example,

$$abcab \in L,\ aaacaaa \in L,\ c \in L,\ bbbacbbba \in L$$

Clearly, it is neither regular nor context-free.

Solution:

```
1. Sweep from start to end.
   If there is no $c$ or there are more than one $c$'s
         REJECT.
   Else
         Move head to start.

2. Read the input character.
   If it is $a$
         Cross it. (Replace by $\times$)
         Move right until $c$.
         Move right over $\times$ characters.
         Read the first character (that is not $\times$).
         If it is $a$
               Cross it.
               Move left until $c$.
               Move left until $\times$.
               Move one step right.
               Go To $2$.
         Else
               REJECT.
   If it is $b$
         ⋮
   //The same idea, but $a \to b$
```

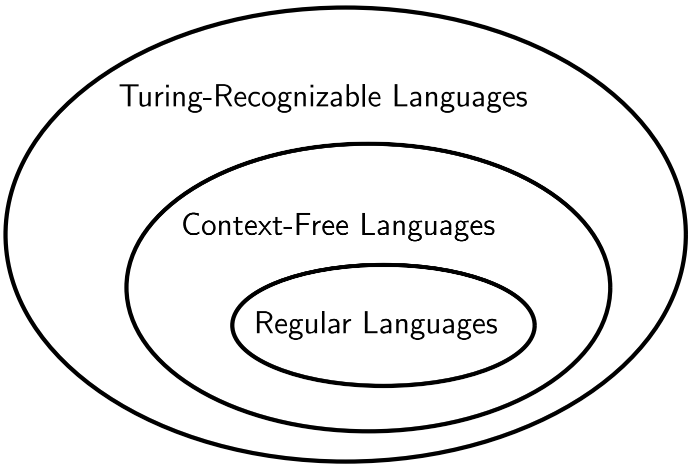
```
//We know that the first character after $\times$'s is neither
$a$ nor $b$, so it must be $c$.

   3. Move right over $\times$ characters.
      Read the first character (that is not $\times$).
      If it is $a$ or $b$
            REJECT.
      Else    //It must be $\square$.
            ACCEPT.
```

We can easily show that this language is not CFL using pumping lemma. But it is clearly Turing-recognizable.

We will later prove that each context-free language is Turing-recognizable.

The tape of the TM is much more powerful than the stack of a PDA.



Example 10–8: Describe a TM that gets n symbols on the tape as input, and gives n^2 symbols as output. The alphabet is $\Sigma = \{a\}$. (This TM is a transducer.)

Solution:

```
1. Move head to start. Search for $a$.
   If found,
         Mark it.      //$(a \to \overline{a})$
         Move head to end. Write $b$ to tape.
         Go To $1$.
   //Write $b$ to $n$ cells.

2. Search for $b$.
   If found
         Delete it.
         Unmark all marked $a$'s.      //$(\overline{a} \to a)$
   Else
         Go To $4$.

3. Move head to start. Search for $a$.
   If found
         Mark it.
         Move head to end.
         Write $c$.
         Go To $3$.
   Else
         Go To $2$.
   //Write $c$ to $n$ cells.

4. Move head to start.
   Delete all marked $a$'s.
   Transform all $c$'s to $a$'s.
   ACCEPT.
```

$\{a,b,c\}^* |$

$\#(a) < \#(b) + \#(c)$

EXERCISES

Describe (in algorithm format) a TM that recognizes the following languages over the alphabet $\Sigma = \{a, b, c\}$ or $\Sigma = \{a, b\}$:

10–1) $L = \big\{a^n\, b^n\, c^n \;\big|\; n, m \geqslant 0\big\}$

10–2) $L = \big\{w \;\big|\; w$ contains equal number of a's, b's and c's.$\big\}$

10–3) $L = \big\{a^n\, b^m\, c^{n+m} \;\big|\; n, m \geqslant 0\big\}$

10–4) $L = \big\{a^n\, b^{n+1}\, c^{4n} \;\big|\; n \geqslant 0\big\}$

10–5) $\big\{w \;\big|$ the number of a's in w is twice the number of c's.$\big\}$

10–6) $L = \big\{w \;\big|\; w = a^k\, b^m\, c^n,$ where $k < m$ and $k < n\big\}$

10–7) $L = \big\{w \;\big|\; w = a^k\, b^m\, c^n,$ where $k < m$ or $k < n\big\}$

10–8) $L = \big\{a^n\, b^m \;\big|\; m = 5n - 3,\; n \geqslant 1\big\}$

10–9) $L = \big\{a^n\, b\, a^{2n}\, b\, a^{3n} \;\big|\; n \geqslant 0\big\}$

10–10) $L = \big\{a^{2n} b^{n+1} \;\big|\; n \geqslant 0\big\}$

10–11) Consider the following languages over $\Sigma = \{0, 1\}$:

$A = \Big\{w \;\Big|\;$ the number of 0's in w is equal to the number of 1's.$\Big\}$

$B = \Big\{w \;\Big|\;$ the number of 0's in w is twice the number of 1's.$\Big\}$

Describe (in algorithm format) a TM that recognizes the language $A \cup B$.

10–12) What language does the following TM recognize?

```
1. Sweep from left to right.
   If there is any $a$ after the first $b$
         REJECT.

2. Move head to start. Search for $a$.
   If found, cross it. (Replace by $\times$)
   Else, Go To $5$.

3. Repeat two times:
         Search for $b$.
         If found
               Cross it.
         Else
               REJECT.

4. Go To $2$.

5. Move head to start. Search for $b$.
   If found
         ACCEPT.
   Else
         REJECT.
```

10–13) What language does the following TM recognize?

```
1. // Input Check.
   If there is no $c$
       REJECT
   If there are two or more $c$'s
       REJECT
   If there is any $a$ after $c$
       REJECT
   If there is any $b$ before $c$
       REJECT

2. Move head to end. Write $b$ to tape.

3. Move head to start. Search for $a$.
   If found
       Cross it. (Replace by $\times$)
   Else
       Go To $6$.

4. Repeat $2$ times:
       Search for $b$.
       If found
           Cross it.
       Else
           REJECT.

5. Go To $3$.

6. Move head to start. Search for $b$.
   If found
       REJECT.
   Else
       ACCEPT.
```

ANSWERS

10–1)

```
1. Sweep from start to end.
   If symbols are out of order (for example, $a$'s after $b$'s)
       REJECT.

2. Move head to start, search for $a$.
   If found
       Cross it. (Replace by $\times$)
   Else
       Go To $5$.

3. Move head to start, search for $b$.
   If found
       Cross it.
   Else
       REJECT.

4. Move head to start, search for $c$.
   If found
       Cross it.
       Go To $2$.
   Else
       REJECT.

5. Move head to start, search for $b$ or $c$.
   If found
       REJECT.
   Else
       ACCEPT.
```

10–2) For a TM, this is essentially the same as previous question. Use the same solution but delete the first part which is about order of symbols.
Note that for NFAs or PDAs, the order of symbols were much more important.

10–3)

```
1. Sweep from start to end.
   If symbols are out of order (for example, $a$'s after $b$'s)
       REJECT.

2. Move head to start.
   Search for $a$ or $b$.
   If found
       Cross it. (Replace by $\times$)
   Else
       Go To $4$.

3. Search for $c$.
   If found
       Cross it.
       Go To $2$.
   Else
       REJECT.

4. Move head to start.
   Search for $c$.
   If found
       REJECT.
   Else
       ACCEPT.
```

10–4)

```
1. Sweep from start to end.
   If any symbol is out of order (for example, $a$'s after $b$'s)
         REJECT.

2. Move head to start. Search for $a$.
   If found
         Cross it. (Replace by $\times$)
   Else
         Go To $6$.

3. Search for $b$.
   If found
         Cross it.
   Else
         REJECT.

4. Repeat four times:
         Search for $c$.
         If found
              Cross it.
         Else
              REJECT.

5. Go To $2$.

6. Sweep from start to end.
   If there is one and only one $b$ and there is no $c$
         ACCEPT.
   Else
         REJECT.
```

10–5)

```
1. If the tape is empty
        ACCEPT.

2. Sweep from start to end. Search for $a$.
   If found
        Cross it.
   Else
        Go To $4$.
   Sweep from start to end. Search for $a$.
   If found
        Cross it.
   Else
        REJECT.

3. Sweep from start to end. Search for $c$.
   If found
        Cross it.
        Go To $2$.
   Else
        REJECT.

4. Sweep from start to end. Search for $c$.
   If found
        REJECT.
   Else
        ACCEPT.
```

10–6)

```
1. Sweep from start to end.
   If symbols are out of order
       REJECT.

2. Move head to start. Search for $a$.
   If found
       Cross it. (Replace by $\times$)
   Else
       Go To $5$.

3. Search for $b$.
   If found
       Cross it.
   Else
       REJECT.

4. Search for $c$.
   If found
       Cross it.
       Go To $2$.
   Else
       REJECT.

5. Move head to start. Search for $b$.
   If found
       Search for $c$.
       If found
           ACCEPT.
       Else
           REJECT.
   Else
       REJECT.
```

10–7)

```
1. ⋯ // Check order as usual.

2. Move head to start. Search for $a$.
   If found, cross it. //Replace by $\times$
   Else, Go To $4$.

3. Search for $b$.
   If found
         Cross it.
         Search for $c$.
         If found
              Cross it.
         Go To $2$.
   Else
         Search for $c$.
         If found
              Cross it.
              Go To $2$.
         Else
              REJECT.

4. Move head to start. Search for $b$.
   If found
         ACCEPT.
   Else
         Search for $c$.
         If found
              ACCEPT.
         Else
              REJECT.
```

10–8)

```
1. If the first input $\neq a$
        REJECT.

2. Sweep from start to end.
   If there is any $a$ after the first $b$
        REJECT.

3. Repeat three times:
        Move head to end.
        Write $b$ to tape.

4. Move head to start. Search for $a$.
   If found
        Cross it.
   Else
        Go To $7$.

5. Repeat five times:
        Search for $b$.
        If found
             Cross it.
        Else
             REJECT.

6. Go To $4$.

7. Move head to start. Search for $b$.
   If found
        REJECT.
   Else
        ACCEPT.
```

10–9)

```
1. Sweep from start to end.
   If there are less than or more than two $b$'s,
         REJECT.

2. Move head to start. Search for $a$ until meeting $b$.
   If found
         Cross it. (Replace by $\times$)
   Else
         Go To $6$.

3. Move head to start. Search for $b$.
   Repeat two times:
         Search for $a$ until meeting $b$.
         If found, cross it.
         Else
              REJECT.

4. Move head to start. Search for $b$. Search for $b$.
   Repeat three times:
         Search for $a$.
         If found, cross it.
         Else
              REJECT.

5. Go To $2$.

6. Move head to start. Search for $a$.
   If found
         REJECT.
   Else
         ACCEPT.
```

10–10)

```
1. Sweep from left to right.
   If there is any $a$ after the first $b$
         REJECT.

2. Move head to start. Search for $b$.
   If found, cross it.
   Else
         REJECT.

3. Move head to start. Search for $a$.
   If found
         Cross it. (Replace by $\times$)
         Search for $a$.
         If found
              Cross it.
         Else
              REJECT.
   Else
         Go To $5$.

4. Search for $b$.
   If found, cross it.
         Go To $3$.
   Else
         REJECT.

5. Move head to start. Search for $b$.
   If found
         REJECT.
   Else
         ACCEPT.
```

10–11)

```
1. // This part checks membership in language $A$.
   Search for $1$.
   If found,
         Mark it.      // $1 \to \overline{1}$
         Search for $0$.
         If found
              Mark it.      // $0 \to \overline{0}$
              Go To $1$.
         Else
              REJECT
   // If $0$'s are fewer than $1$'s, the string belongs to
   // neither $A$ nor $B$.
   Else
         Search for $0$.
         If found
              Go To $2$.
         Else
              ACCEPT.

2. // Unmark all marked symbols, in other words, RESET.
   Search for $\overline{0}$.
   If found,
         Unmark it.      // $\overline{0} \to 0$
         Go To $2$.

3. Search for $\overline{1}$.
   If found,
         Unmark it.      // $\overline{1} \to 1$
         Go To $3$.
```

```
4. // This part checks membership in language $B$.
   Search for $1$.
   If found,
         Mark it.
         Repeat two times:
              Search for $0$.
              If found
                   Mark it.
              Else
                   REJECT.
         Go To $4$.
   Else
         Search for $0$.
         If found
              REJECT.
         Else
              ACCEPT.
```

10–12) $L = \{a^n b^m \mid m \geqslant 2n+1\}$

10–13) $L = \{a^n \, c \, b^{2n-1} \mid n \geqslant 1\}$

Week 11

Turing Machines - II

11.1 Variants of Turing Machines

There are different versions of TMs but they are computationally equivalent, in other words, they recognize the same languages.

Doubly Infinite Tape: A TM with a tape infinite in both directions can be simulated on a semi-infinite tape.

$\cdots$			a	b	c	d	e	f			$\cdots$

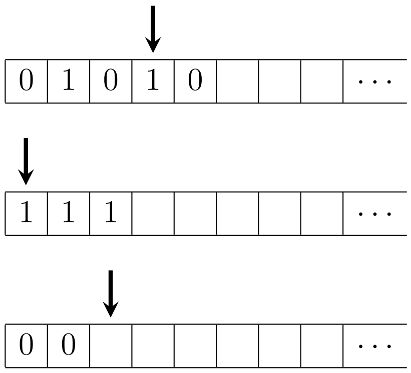
We will divide the tape in two, use new special symbols and write the input as follows:

$\$$	d	e	f	$\#$	a	b	c	$\%$			$\cdots$

If the TM reads $\$$ at any state, it will enter a special subroutine. The head will go right all the way to $\%$, then move one unit left and return to same state.

Meeting $\#$ will be like meeting $\square$. This symbol will move as the input expands and contracts. Other details are left as an exercise.

Multitape Turing Machines: A multitape Turing machine has several tapes. Each tape has its own head for reading and writing. Initially, the input appears on tape 1, and the others start out blank. This machine can also be simulated by a standard TM with semi-infinite tape.



This time, we have to keep in memory all the data and also all the head positions. For this, we can define a new version of each tape symbol, for example $\overline{0}, \overline{1}, \overline{\square}$ etc. These symbols will mark the placement of the head. We also need a separator symbol $\#$.

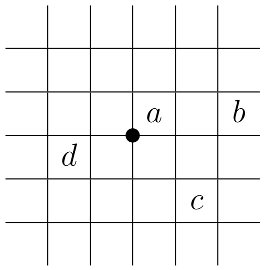
With this arrangement, the information on the 3 tapes above will be put on a single tape as:

0	1	0	$\overline{1}$	0	$\#$	$\overline{1}$	1	1	$\#$	0	0	$\overline{\square}$		$\cdots$

We can think of it is as if our single tape machine had 3 virtual tapes.

While these machines are equivalent in computational power, they are not equivalent in computational efficiency. For example, consider the language $a^nb^nc^nd^n$ and a $4-$tape machine. The input can be copied on all tapes and separate heads can keep track of separate symbols rather than going back and forth.

Two-Dimensional Tape: Consider a TM that uses the whole plane as memory. That is equivalent to infinitely many tapes:



But again, it is possible to simulate this machine with an ordinary TM, so it is not more powerful. If we can use just three tapes, we can write the symbol, its $x-$ and $y-$ coordinates:

a	b	c	d			$\cdots$

1	#	3	#	2	#	-2	#		$\cdots$

1	#	1	#	-2	#	-1	#		$\cdots$

We have previously seen that a single tape and three tape machines are equivalent.

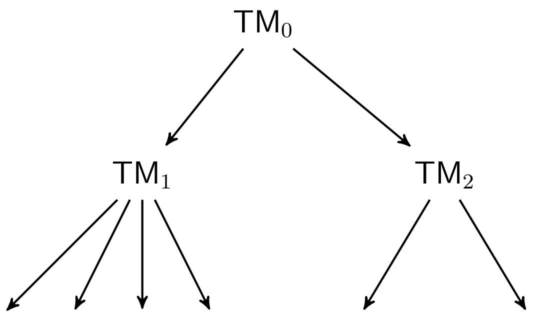
Again, there are many details we have to think about during implementation. For example, 3 could be represented as 111. For the minus sign, we need another special symbol.

The multidimensional TM will find the neighbor of a cell easily, by going one unit left, right, up, or down. The $3-$tape machine must search through the whole tape for correct coordinates.

But all these details about efficiency are unimportant from our current point of view. These machines recognize the same languages.

Nondeterministic Turing Machines: Up to now, we have been working with deterministic Turing Machines. What happens if we include nondeterminism?

A nondeterministic TM may have two or more outputs for the same input. In other words, its computation branches at each nondeterministic choice. (We can imagine it as a large number of machines working in parallel and generating new machines.)



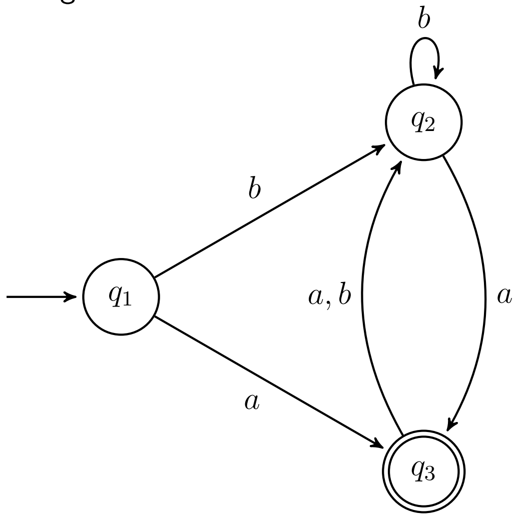
An ordinary TM can simulate this computation. It must store all the contents of the tape, the state of the machine, and the position of the head. We can work with a $3-$tape deterministic TM, but a single tape is also sufficient. At each branching, we must copy the whole contents of the new machines.

If the nondeterministic TM accepts, this result is somewhere in the tree. Our equivalent deterministic TM will search the tree and halt when it finds accept.

When searching the tree, we have to use breadth-first search, not depth-first search. Otherwise, we may go down an infinitely long branch and fail to find the accept state although it existed.

11.2 Universal Turing Machine

It is possible to represent a DFA with a string. For example, the following state diagram



could be encoded as:

#	q	1	R	a	q	3	b	q	2	#	q	2	R	a	q	3	$\cdots$

$\cdots$	b	q	2	#	q	3	A	a	q	2	b	q	2				

Can you see the correspondence between these two?

A diagram is easier for human eyes but computers prefer strings. Actually, the string above is also designed for humans. For a computer, we would use only 0's and 1's. We can number states as $1, 11, 111$, use 0's as cell separators, give a special number to each symbol like A, R, a, b, use 00 for the separator # etc. Then:

$$\begin{aligned} \#q1Raq3bq2 &\rightarrow 0010101011101101 1 \\ \#q3Aaq2bq2 &\rightarrow 0011101101011011011 \end{aligned}$$

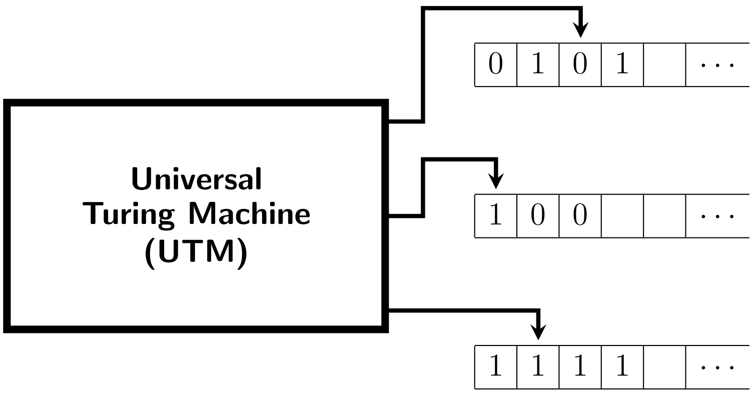
Such implementation details are not important for our purposes.

Similar to the previous example, we can represent NFAs, PDAs, or TMs with strings. They all have finitely many states and therefore finite amount of information. Once we agree on a convention, they can all be encoded with binary strings.

This allows us to define a **Universal Turing Machine (UTM)**.

The TMs we have seen up to now were like computer programs that accomplished a certain job. But a Universal Turing Machine is just like a computer. It can execute any program stored in the form of a TM.

Universal Turing Machine has 3 tapes: One for the TM it will simulate, one for the input, and another to keep the current state in memory.



Our UTM reads input on tape 2 and the state of the given TM on tape 3. Then, based on the description of the given TM on tape 1, it will continue the computation.

During the computation, the contents of tapes 2 and 3 will change. Tape 1 is fixed. At the end of the computation, the output will be on tape 2.

The UTM can simulate any TM, in other words, any computer!

11.3 Decidable Languages

Turing Machines do not always give the result *ACCEPT* or *REJECT*. Sometimes they continue the computation indefinitely, in other words, they *LOOP*. This is a situation we have not encountered for DFAs or PDAs.

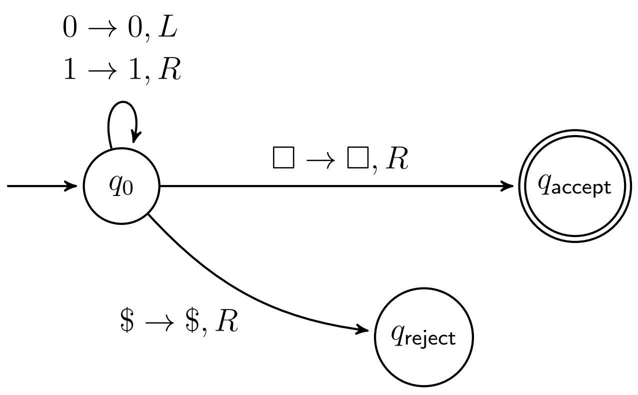
For example, consider the TM:

```
1. If input is $1$
        Move head right, Go To $1$.

2. If input is $\square$
        ACCEPT.

3. If input is $0$
        If head is not at start
            Move head left, Go To $1$.
        Else
            REJECT.
```

A low-level representation would be:



(Here, $\$$ indicates the left end of the tape.)

Clearly, this TM accepts 1111 and rejects 0011. What about 1100?

We have to make a distinction between TMs that may loop and TMs that always give an answer in finitely many steps:

A Turing Machine that halts on all inputs is called a **decider**. A language L is called a **Turing-decidable language** if there exists a TM for that language that is a decider.

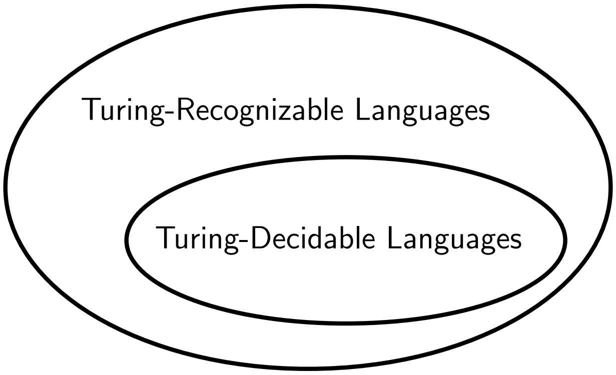
In other words, if M is a decider for L, then:

$$w \in L \;\Rightarrow\; M \text{ ends in } q_{\text{accept}}$$
$$w \notin L \;\Rightarrow\; M \text{ ends in } q_{\text{reject}}$$

If M is a recognizer for L, then:

$$w \in L \;\Rightarrow\; M \text{ ends in } q_{\text{accept}}$$
$$w \notin L \;\Rightarrow\; M \text{ ends in } q_{\text{reject}} \text{ or } M \text{ loops}$$

Using a recognizer in practice, a very long computation would put us in a dilemma:
Is it going to accept if we wait long enough, or did it loop?



Some books use the alternative terminology:

Turing-Recognizable	$\longleftrightarrow$	**Recursively Enumerable**
Turing-Decidable	$\longleftrightarrow$	**Recursive**

There are different methods to describe a Turing Machine:

1. Low-level description using a diagram similar to PDAs.
2. Medium-level description using pseudo-code.
3. High-level description.

The lower the level, the more complicated the description. Up to now, we used the first two levels. From now on, we will use a high-level description for TMs, because our real concern is algorithms and TM is just a model for an algorithm.

Example 11–1: There are two computer programs. They take a finite page of text as input and produce another page of text as output. A page is defined as n characters, where n is a fixed number. We think of them as black boxes. (We can't see what's inside.)

A Turing machine can give any input it wants to these two programs. It will ACCEPT if they always return the same output for the same input, REJECT otherwise.

Is this TM a decider?

Solution: Yes, it is a decider. The TM can check all possible inputs because there are finitely many inputs.

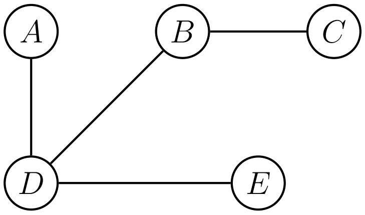
Supposing there are k different characters and n characters per page, it needs to check

$$n^k$$

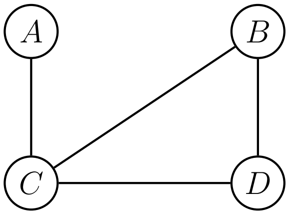
different inputs for an ACCEPT. After that, it will halt.

Of course, it may REJECT much earlier than that.

Example 11–2: Let L be the language consisting of all strings representing trees. A **tree** is an undirected connected graph that does not contain any cycles.



Tree



Not a Tree

For example, the first graph can be represented by the string

$$A00D\,000B00C0D\,000C00B\,000D00A0B0E\,000E00D$$

Alternatively, we can use the notation $A \to 1, \quad B \to 11$, etc. These are just examples. We can use many other representations.

Can we find a TM that decides L? In other words, is L decidable?

Solution: Yes, the following TM always halts:

```
1. Starting from any vertex, use DFS or BFS to reach
   other vertices.

2. If there is an unreached vertex
         REJECT.

3. Use DFS or BFS to detect cycles.

4. If there is a cycle
         REJECT.
     Else
         ACCEPT.
```

Note that this is a high-level description.

11.4 Acceptance and Emptiness Problems

We have seen that we could represent DFAs by strings. Now we will consider sets of DFAs as languages. We will use the notation $\langle O \rangle$ to denote the string representation of the object O.

Example 11–3: Is the following language decidable?

$$\mathsf{A}_{\mathsf{DFA}} = \left\{ \langle D, w \rangle \;\middle|\; D \text{ is a DFA that accepts the string } w \right\}$$

In other words, given a deterministic finite automaton D and a string w, can a TM determine if w is accepted by D or not in finite time?

Solution: A DFA specifies what to do at each state for each input. We can express this as an algorithm, therefore we can also design a Turing machine that simulates a DFA.

This TM will store a description of DFA and the input string on different parts of the tape (or on different tapes if we use a multi-tape machine) and simulate the DFA.

It will also store information about the position on the input and the current state.

```
1. Simulate $D$ with input $w$.

2. When the input ends, check the current state.
   If it is an accept state
         ACCEPT
    Else
         REJECT
```

This algorithm never loops, so $\mathsf{A}_{\mathsf{DFA}}$ is decidable.

Example 11–4: Consider the language:

$$\text{A}_{\text{CFG}} = \big\{\langle G, w\rangle \;\big|\; G \text{ is a CFG that generates the string } w\big\}$$

Is A_{CFG} decidable?

In other words, given a context-free grammar G and a string w, can a TM determine if $w \in L(G)$ or not in finite time?

Solution: One idea is to generate all possible strings using G and then compare each to w, accept if they are identical. But this algorithm may or may not halt. (It is like searching for a node in an infinite tree using DFS.)
A better idea is to transform the grammar to Chomsky normal form. Then:

```
1. If $w = \varepsilon$
     Check all derivations with $1$ step.
         If $w$ is obtained
             ACCEPT.
2. Let $n = |w|$.
3. Check all derivations of $G$ with exactly $2n - 1$ steps.
         If $w$ is obtained
             ACCEPT
4. REJECT
```

We can generalize the last two examples to NFAs, regular expressions, and PDAs because we can convert these to equivalent DFAs or CFGs. So all of these languages are decidable.

Example 11–5: Consider the language:

$$\mathsf{E}_{\mathsf{DFA}} = \{\langle D\rangle \mid D \text{ is a DFA and } L(D) = \varnothing\}$$

Is $\mathsf{E}_{\mathsf{DFA}}$ decidable?

In other words, given a deterministic finite automaton D, can a TM determine if there is any string accepted by D or not in finite time?

Solution: This time we have no specific string. We have to start from the start state and try to reach an accept state.

```
1. Mark the start state.

2. Repeat until no new states get marked:
       Check all marked states.
           If there is a transition leading to an
           unmarked state
               Mark it.

3. Check all marked states.
       If any of them is an accept state
           REJECT.

4. ACCEPT.
```

Example 11–6: Consider the language:

$$\mathsf{E}_{\mathsf{CFG}} = \left\{ \langle G \rangle \;\middle|\; G \text{ is a CFG and } L(G) = \varnothing \right\}$$

Is $\mathsf{E}_{\mathsf{CFG}}$ decidable?

In other words, given a context-free grammar G, can a TM determine if G generates any string or not in finite time?

Solution: Once again, using CNF for the grammar simplifies things. We will try to construct a path from the terminals to the start variable:

```
1. Mark all terminal symbols.
     If there is no terminal
          ACCEPT.

2. Repeat until no new variables get marked:
          Check all the rules.
               If there is a rule $A \to a$,
                    Mark $A$.
               If there is a rule $A \to BC$
                    If both $B$ and $C$ are marked
                         Mark $A$.

3. If $S$ is not marked
          ACCEPT.
     else
          REJECT.
```

Example 11–7: Consider the language:

$$L = \{\langle D, k \rangle \mid D \text{ is a DFA and it rejects all strings } w \text{ if } |w| \leqslant k\}$$

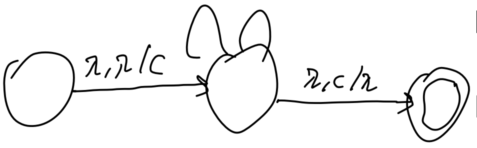
Is L decidable?

In other words, given a deterministic finite automaton D and an integer k, can a TM determine if D accepts any string of length less than or equal to k?

Solution: Once again we will use a TM as a simulator:

$S \to AS$

```
1. Mark start state.
2. Repeat $k$ times:
       Check all states that can be reached from
       marked states in one step.
           If it is an accept state
               REJECT.
           Else
               Mark it.
3. ACCEPT.
```



20

20

10

Another idea is:

- Make a list of all strings of length $\leqslant k$,
- Check them one by one using simulation,
- REJECT if any string is accepted.

That is bad style but still works. (We will be concerned with the number of steps a program needs in the last two chapters.)

We can ask similar questions about a TM. Remember that we can encode any TM by a string, and we have a Universal Turing Machine (UTM) that can simulate any given TM on any given string.

This is a very good model of computation, where UTM is like a computer, given TM is like a program and the given string is the data of the program.

Please keep in mind that all the languages we have seen up to this point are Turing-decidable.

Example 11–8: Consider the language:

$$\mathsf{A}_{\mathsf{TM}} = \big\{ \langle M, w \rangle \;\big|\; M \text{ is a TM that accepts the string } w \big\}$$

Is A_{TM} decidable?

In other words, given a Turing Machine M and a string w, can UTM determine if w is accepted by M or not in finite time?

Example 11–9: Consider the language:

$$\mathsf{E}_{\mathsf{TM}} = \big\{ \langle M \rangle \;\big|\; M \text{ is a TM and } L(M) = \varnothing. \big\}$$

Is E_{TM} decidable?

In other words, given a Turing Machine M, can UTM determine (after finitely many steps) if M rejects all strings or accepts at least one string?
Unfortunately, we cannot answer these questions at this stage. Unlike DFAs or PDAs, TMs can enter infinite loops. Therefore we cannot be sure that these languages are decidable.

Please think about these two questions before reading next chapter. You will understand the potentials and limitations of TMs (and therefore computers) better.

EXERCISES

11–1) Consider a TM that can not delete or modify any symbol it has written to tape. The tape is still infinite but it is write-once. Is this equivalent to a standard TM?

11–2) Consider a PDA that has two stacks. Is this equivalent to a standard TM?

11–3) Consider a TM M that recognizes a language L. It enters a loop for some strings which are not in L. Can we claim L is not decidable?

11–4) Consider the language L over $\Sigma = \{0, 1\}$ consisting of strings that are binary representations of prime numbers. For example $101 \in L,\ 110 \notin L,\ 111 \in L$ etc. Is L decidable?

11–5) If there is a state in a DFA that is not reachable from the start state, it is redundant. Consider the set of all DFAs that do not contain any redundant state. Think of it as a language. Is this language decidable?

11–6) Is the complement of a decidable language decidable?

11–7) Is the union of two decidable languages decidable?

11–8) Is the intersection of two decidable languages decidable?

11–9) Is the concatenation of two decidable languages decidable?

11–10) Is the star of a decidable language decidable?

11–11) Is the complement of a Turing-Recognizable language Turing-Recognizable?

ANSWERS

11–1) Yes. At each step, it has to copy and paste all contents of the tape, modifying a single cell. That's very inefficient but it works.

11–2) Yes. If we start by storing the tape of a TM in one of the stacks and leave the other empty, we can reach any part of the tape by transferring symbols from one stack to the other.

11–3) No. We can claim this only if all TMs for L loop.

11–4) Yes. We can write an algorithm that tests whether the given number is prime or not. Then, we can express this algorithm in TM format. It will be complicated but it will always halt. It won't loop for any input.

Actually, if you check the TMs we have given in the last chapter, you will realize that all of them are deciders without exception. We haven't seen any TM that is a recognizer but not a decider.

11–5) Yes.

1. Mark start state. Repeat:
2. Mark all states reachable from marked states.
3. If no new states are marked, search for an unmarked state.
4. If there is an unmarked state, REJECT.
   Otherwise, ACCEPT

11–6) Yes. If L is decidable then $\overline{L}$ is also decidable. Just swap the REJECT and ACCEPT states of the given TM deciding L.

11–7) Yes. If L_1 and L_2 are decidable then $L_1 \cup L_2$ is also decidable. To design a TM, call the TMs for L_1 and L_2 as subroutines and connect their results with OR operation.

11–8) Yes. If L_1 and L_2 are decidable then $L_1 \cap L_2$ is also decidable. To design a TM, call the TMs for L_1 and L_2 as subroutines and connect their results with AND operation.

11–9) Yes. If L_1 and L_2 are decidable then $L_1 \circ L_2$ is also decidable. TM is more complicated this time. We should divide the given input into two parts at all possible points, then give these inputs to the TMs for L_1 and L_2 and accept if both accept. If there is no such division, we reject.

This process may take a long time but it always ends after finitely many steps. In other words, it never loops.

11–10) Yes. If L is decidable then L^* is also decidable. Construction of TM is similar to $L_1 \circ L_2$ given in previous answer.

11–11) No. Suppose L is Turing-Recognizable and the string $s \notin L$ (in other words $s \in \overline{L}$.) If the given TM for L rejects s that's OK. But perhaps it loops!

There is no way to we can be sure there is a recognizer for $\overline{L}$ in general.

Week 12

An Undecidable Language

12.1 Uncountable Sets and Diagonalization

Before we return back to languages that are recognizable but not decidable, we need to learn more about infinity. We can easily compare the sizes of two finite sets. What about infinite ones?

In other words, given two infinite sets, can we say one is *"larger"* than the other in some sense, or are they always the same? Is there an infinity greater than another infinity?

For example, consider the sets:

$$\begin{array}{rrcl} \text{All Integers:} & \mathbb{Z} & = & \{\ldots,-2,-1,0,1,2,\ldots\} \\ \text{Positive Integers:} & \mathbb{Z}^+ & = & \{1,2,3,\ldots\} \\ \text{Even Positive Integers:} & 2\mathbb{Z}^+ & = & \{2,4,6,\ldots\} \\ \text{Prime Numbers:} & \mathbb{P} & = & \{2,3,5,7,11,\ldots\} \end{array}$$

How can we compare their sizes? Which one is the largest?

Perhaps we should consider two infinite sets to be equivalent in size if there is a one-to-one correspondence (a $1-1$ onto function) between them. This is the same idea we use for finite sets.

A set S has **cardinality** n if we can find a one-to-one correspondence between the sets $\{1, 2, \ldots, n\}$ and S. The cardinality of the empty set is defined to be 0.

If it is impossible to find a one-to-one correspondence between a finite set and the non-empty set S, then S is called an infinite set.

Two sets have the same cardinality if we can find a one-to-one correspondence between them.

A set S is called **countable** if it is finite or if it has the same cardinality as $\mathbb{Z}^+$. It is called **uncountable** if it is not countable.

For example, positive integers and positive even integers have the same cardinality, because we can find a one-to-one correspondence between them:

$$\begin{array}{ccc} 1 & \longleftrightarrow & 2 \\ 2 & \longleftrightarrow & 4 \\ 3 & \longleftrightarrow & 6 \\ \vdots & & \vdots \end{array}$$

We can express this as a $1-1$ and onto function:

$$f : \mathbb{Z}^+ \to 2\mathbb{Z}^+, \qquad f(n) = 2n$$

We can do the same for prime numbers, but it is more difficult to find f explicitly.

$$\begin{array}{ccc} 1 & \longleftrightarrow & 2 \\ 2 & \longleftrightarrow & 3 \\ 3 & \longleftrightarrow & 5 \\ 4 & \longleftrightarrow & 7 \\ \vdots & & \vdots \end{array}$$

Example 12–1: Is $\mathbb{Z}$ countable?

Solution: At first, $\mathbb{Z}$ seems to be a larger set, with more than twice the elements of $\mathbb{Z}^+$, but actually, they have the same cardinality.

We can do the trick of matching odd positive integers with positive integers, and the evens with zero and negative integers:

$$\begin{array}{ccr} 1 & \longleftrightarrow & 1 \\ 2 & \longleftrightarrow & 0 \\ 3 & \longleftrightarrow & 2 \\ 4 & \longleftrightarrow & -1 \\ 5 & \longleftrightarrow & 3 \\ 6 & \longleftrightarrow & -2 \\ \vdots & & \vdots \end{array}$$

In function format, this is:

$$f : \mathbb{Z}^+ \to \mathbb{Z}, \qquad f(n) = \begin{cases} \dfrac{n+1}{2} & \text{if } n \text{ is odd} \\[2ex] 1 - \dfrac{n}{2} & \text{if } n \text{ is even} \end{cases}$$

This function is $1-1$ and onto. Its inverse is:

$$f^{-1} : \mathbb{Z} \to \mathbb{Z}^+, \qquad f^{-1}(n) = \begin{cases} 2n-1 & \text{if } n > 0 \\[2ex] 2(1-n) & \text{if } n \leqslant 0 \end{cases}$$

So $\mathbb{Z}$ is countable and it has the same cardinality as $\mathbb{Z}^+$.

Example 12–2: If we add a finite number of elements to a countable set, will the resulting set be countable?

Solution: Yes. If S_1 is countable, there is a $1-1$ onto function f_1 from $\mathbb{Z}^+$ to S_1. Suppose we add n new elements to obtain S_2. Then, we can slightly modify the function and obtain another $1-1$ onto function f_2 as follows:

$$\begin{array}{ccc} & f_1 & \\ 1 & \longleftrightarrow & s_1 \\ 2 & \longleftrightarrow & s_2 \\ 3 & \longleftrightarrow & s_3 \\ \vdots & & \vdots \end{array} \quad \Rightarrow \quad \begin{array}{ccc} & f_2 & \\ 1 & \longleftrightarrow & a_1 \\ 2 & \longleftrightarrow & a_2 \\ \vdots & & \vdots \\ n & \longleftrightarrow & a_n \\ n+1 & \longleftrightarrow & s_1 \\ n+2 & \longleftrightarrow & s_2 \\ \vdots & & \vdots \end{array}$$

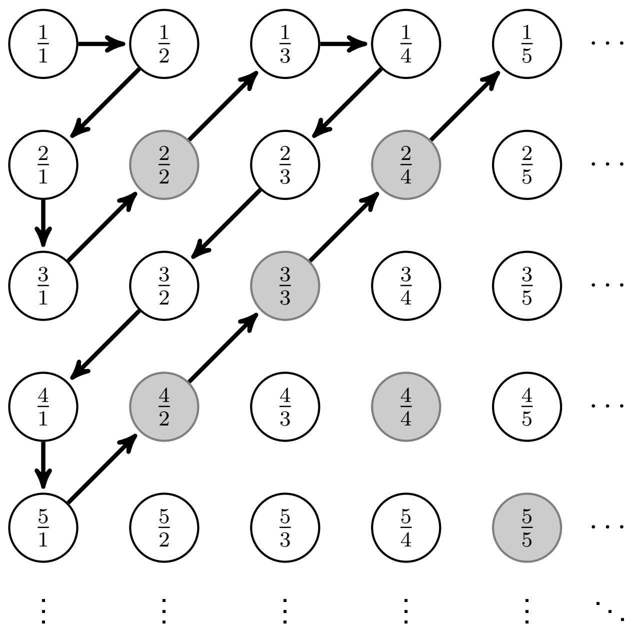
Example 12–3: If S and T are countable sets, is $S \cup T$ also countable?

Solution: Yes. We can obtain the $1-1$ onto function as follows:

$$\begin{array}{ccc} 1 & \longleftrightarrow & s_1 \\ 2 & \longleftrightarrow & t_1 \\ 3 & \longleftrightarrow & s_2 \\ 4 & \longleftrightarrow & t_2 \\ \vdots & & \vdots \end{array}$$

Example 12–4: Is $\mathbb{Q}$ countable?

Solution: If we can count positive rationals, we can also count the negative ones. Consider this infinite matrix that contains all positive rationals. (All of them multiple times.)



If we count (start making a list) using rows or columns, we will be stuck. But if we count using diagonals, we can count all of them.

We have to skip over the ones we already counted. For example, $\frac{2}{4}$ is the same number as $\frac{1}{2}$, therefore $\frac{2}{4}$ won't be on the list.

Expressing this function using formulas is complicated. But in principle, the matching can be done.

So $\mathbb{Q}$ is countable.

Example 12–5: Is $\mathbb{R}$ countable?

Solution: We will use Cantor's diagonal argument.

For our purposes *"countable"* and *"listable"* have the same meaning. If there is a $1-1$ onto function from $\mathbb{Z}^+$ to the given set, there is a first element, second element, etc., and this list contains all elements of the set.

Suppose real numbers on $\big[0,\,1\big]$ are countable, and a list of them is given. For example, suppose the first number on the list is $a_1 = 0.36541\ldots$, second number is $a_2 = 0.87523\ldots$ etc.

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & \cdots \\ \hline a_1 & \boxed{3} & 6 & 5 & 4 & 1 & \cdots \\ a_2 & 8 & \boxed{7} & 5 & 2 & 3 & \cdots \\ a_3 & 6 & 1 & \boxed{1} & 5 & 2 & \cdots \\ a_4 & 8 & 2 & 6 & \boxed{9} & 4 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

Now, we can find a number that is not in the list by checking diagonal digits and choosing a different one each time:

$$a = 0.5867\ldots$$

Clearly, $a \neq a_1$ because their first digits are different. Also $a \neq a_2$ because their second digits are different. This is the case for all digits, in other words, although $a \in \big[0,\,1\big]$, a is not on the list!

Therefore the initial assumption that there is a list of all real numbers on $\big[0,\,1\big]$ must be wrong.

In other words, $\mathbb{R}$ is uncountable.

Example 12–6: Is the set of all binary sequences countable? (A binary sequence is an infinite sequence composed of 0's and 1's.)

Solution: We can use the same argument in the last question.

- Assume the given set is countable.
- Make a list of them.
- Generate an element not on the list using the diagonal.

For example, suppose the first binary sequence in our (hypothetical) list is $b_1 = \{0, 1, 0, 1, 0, 1, \ldots\}$, the second one is $b_2 = \{1, 1, 1, 0, 0, 0, \ldots\}$ etc.

$$\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & \cdots \\
\hline
b_1 & \boxed{0} & 1 & 0 & 1 & 0 & 1 & \cdots \\
b_2 & 1 & \boxed{1} & 1 & 0 & 0 & 0 & \cdots \\
b_3 & 0 & 0 & \boxed{1} & 1 & 0 & 0 & \cdots \\
b_4 & 0 & 1 & 0 & \boxed{0} & 1 & 0 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

Now generate b by checking i$^{\text{th}}$ digit of i$^{\text{th}}$ sequence:

- If it is 0, choose 1,
- If it is 1, choose 0.

$$b = \{1, 0, 0, 1, \ldots\}$$

Clearly, b is not on the list so the set of binary sequences is uncountable.

(If we use binary notation for real numbers, this question is identical to the previous one.)

Example 12–7: Is the set of all functions $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ countable?

Solution: NO. We can think of each of these functions as a sequence of numbers. If we order these sequences as columns, we obtain an infinite matrix similar to that of $\mathbb{Q}$. Suppose the first function is:

$$\begin{array}{ccr}
\mathbb{Z}^+ & \xrightarrow{f_1} & \mathbb{Z}^+ \\
\hline
1 & \to & 19 \\
2 & \to & 5 \\
3 & \to & 127 \\
4 & \to & 58 \\
\vdots & & \vdots
\end{array}$$

Then the matrix looks like

$$\begin{array}{c|rrrrc}
 & f_1 & f_2 & f_3 & f_4 & \cdots \\
\hline
1 & \boxed{19} & 834 & 11 & 1 & \cdots \\
2 & 5 & \boxed{7} & 2 & 10 & \cdots \\
3 & 127 & 9 & \boxed{3} & 44 & \cdots \\
4 & 58 & 13 & 8 & \boxed{49} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

Once again we can construct an element that is not on the list using the diagonal. That will be the function f such that

$$\begin{aligned}
f(1) &\neq 19 \\
f(2) &\neq 7
\end{aligned}$$

etc.

12.2 Turing-Unrecognizable Languages

Let's remember that using some convention, we can represent any Turing Machine or computer program, or any algorithmic method of solution of a problem as a binary string.

An algorithm may have more than one implementation, so it can be represented by several strings. That's not important.

Also, whatever the convention we use for representation, there will be some strings that do not represent any TM, just as it is possible to type some statements causing syntax errors for any given computer language. That's not very important either. We will either disregard them or consider them as the simplest TM that immediately halts and rejects all inputs. Their language will be $\varnothing$.

Now, based on this correspondence, imagine the set of all possible TMs. Is this set countable?

Don't forget that while each TM can be arbitrarily large, it must have finitely many states. That means the answer is "*Yes*". The set of all finite binary strings is countable. Here is a list of all of them:

$$\begin{array}{r} 0 \\ 1 \\ 00 \\ 01 \\ 10 \\ 11 \\ 000 \\ 001 \\ 010 \\ \vdots \end{array}$$

Next question:

Consider the set of all languages on $\Sigma = \{0, 1\}$. Is this set countable?

We can easily show that the answer is "*No*" using the same diagonalization idea. In other words, assume the set is countable, make a list of them. Construct a matrix where rows are languages, columns are finite strings and matrix elements are 0 or 1, denoting membership in the language. Finally, construct a language not on the list.

That means there are some languages that are not recognizable by any TM!

Alternatively, consider a function, any function

$$f : \mathbb{Z}^+ \to \mathbb{Z}^+$$

Is it possible to write a computer program that computes it? (That means give the correct output for all inputs.)

If the function is $f(x) = 2x + 1$ or $f(x) = x^3$ the answer is obviously "*Yes*". We are considering the set of all possible functions from positive integers to positive integers. This set is uncountable, but the set of all computer programs is countable. So there must be some functions that are NOT computable by any computer program!

In both cases, there are too many problems and too few solutions. Some problems must be unsolvable.

We have proved that there must be some languages that are not Turing-recognizable, but we still do not have any specific examples. Next, we will construct such a language explicitly.

The Diagonal Language:

The diagonal language $\mathsf{L_D}$ is defined as the set of all TM representations that are not accepted by themselves.

$$\mathsf{L_D} = \{\langle M\rangle \mid \langle M\rangle \notin L(M)\}$$

For example, consider a TM that accepts all binary strings of length 500 or less.

If the implementation of this TM is a string of length > 500, this implementation is in $\mathsf{L_D}$.

Another example is a TM that checks the length of the input string w modulo 3.

- If $|w| \equiv 0 \pmod 3$ it accepts.
- If $|w| \equiv 1 \pmod 3$ it rejects.
- If $|w| \equiv 2 \pmod 3$ it enters an infinite loop.

Again, this algorithm can be implemented in many different ways. Let $\langle M\rangle$ be the binary string corresponding to one specific implementation.

If $\big|\langle M\rangle\big| \equiv 1 \pmod 3$ or $\big|\langle M\rangle\big| \equiv 2 \pmod 3$ then $\langle M\rangle \in \mathsf{L_D}$.

Now let's check the general case. The set of all finite binary strings will be considered as TM implementations. This set is countable. Let's make a list of them and give all these strings as inputs to all the machines. This will result in an infinite table.

The question is:

"Does the i^th^ *machine accept the* j^th^ *input?"*

If the machine accepts, the answer is YES. If it rejects or loops, the answer is NO.

$$\begin{array}{rc|cccc} & & \multicolumn{4}{c}{\text{Input Strings}} \\ & & \langle M_1\rangle & \langle M_2\rangle & \langle M_3\rangle & \cdots \\ \hline & M_1 & \boxed{\text{No}} & \text{Yes} & \text{No} & \cdots \\ \text{Turing Machines} & M_2 & \text{Yes} & \boxed{\text{Yes}} & \text{No} & \cdots \\ & M_3 & \text{No} & \text{No} & \boxed{\text{Yes}} & \cdots \\ & \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

Based on the definition, we can easily see that $M_1 \in$ L$_\text{D}$ but $M_2 \notin$ L$_\text{D}$ and $M_3 \notin$ L$_\text{D}$. If L$_\text{D}$ was in the table, its row would be:

$$\begin{array}{c|cccc} \text{L}_\text{D} & \text{Yes} & \text{No} & \text{No} & \cdots \end{array}$$

But this is impossible!

The *"Yes"* answers of each row constitute a subset of $\{0,1\}^*$, in other words, a language. Similar to Cantor's diagonal counterexample, this language L$_\text{D}$ is different from all the languages in the table. It cannot be one of the rows.

In other words, the language L$_\text{D}$ is not recognizable by any TM.

If L$_\text{D}$ accepts $\langle$L$_\text{D}\rangle$ then it must reject $\langle$L$_\text{D}\rangle$.
If L$_\text{D}$ does not accept $\langle$L$_\text{D}\rangle$ then it must accept $\langle$L$_\text{D}\rangle$.

This is similar to the following logical paradoxes:

- This sentence is wrong.
- Everything I am telling is a lie.

12.3 $\mathsf{A_{TM}}$ and $\mathsf{HALT_{TM}}$

In this section, we will give an example of a language that is Turing-recognizable but not Turing-decidable.
We can construct TMs that decide simpler questions, for example, about a DFA accepting a given string. Now consider a larger class. Define:

$$\mathsf{A_{TM}} = \big\{ \langle M, w \rangle \bigm| M \text{ is a TM and } M \text{ accepts } w. \big\}$$

Is this decidable? In other words, can we find a TM that answers the following question after finitely many operations?

Is the given Turing Machine going to accept the given string?

Consider the following TM:

1. Simulate M on input w.
2. If M enters its accept state
   ACCEPT.
3. If M enters its reject state
   REJECT.

This is a Universal Turing Machine (UTM) because it can simulate any other Turing machine using the description of that machine. But it is not a decider because it may enter an infinite loop. In other words, UTM *recognizes* $\mathsf{A_{TM}}$, it does not *decide* $\mathsf{A_{TM}}$.

Let's assume, for the sake of argument, that $\mathsf{A_{TM}}$ is decidable. This means, there is a TM deciding it. Let's call it TM01.

$$\text{TM01}\big(\langle M, w \rangle\big) = \begin{cases} \text{ACCEPT} & \text{if } M \text{ accepts } w. \\ \text{REJECT} & \text{if } M \text{ does not accept } w. \end{cases}$$

Here, "does not accept" means rejects or loops. The machine M might loop but the machine TM01 does not. It is a decider.

Now construct a new Turing machine TM02. This new TM calls TM01 as a subroutine and gives it the input $\langle M, \langle M\rangle\rangle$. This means M as a TM and $\langle M\rangle$ as a string.

TM02 accepts all Turing Machines that do not accept themselves. (This is identical to language $\mathsf{L_D}$ described in the previous section.)

$$\text{TM02}(\langle M\rangle) = \begin{cases} \text{ACCEPT} & \text{if } M \text{ does not accept } \langle M\rangle \\ \text{REJECT} & \text{if } M \text{ accepts } \langle M\rangle \end{cases}$$

Note the distinction between M and $\langle M\rangle$. One is like a program as a set of instructions and the other is the same program simply as a text file. The idea is to give the text of the source code of the program to the same program as input.

What happens when we run TM02 with itself $\langle$TM02$\rangle$ as input? In that case, we get a contradiction. This means that TM02 and therefore TM01 cannot exist. In other words, $\mathsf{A_{TM}}$ is not decidable.

We have already seen that there is no TM for $\mathsf{L_D}$.

A similar paradox exists in set theory:
Let R be the set of all sets that are not elements of themselves.

$$R = \{S \mid S \notin S\}$$

Is R an element of itself?

- Suppose $R \in R$. But that means $R \notin R$.
- Suppose $R \notin R$. But that means $R \in R$.

Contradiction!

Halting Problem: Another language that is Turing-recognizable but not Turing-decidable is $\mathsf{HALT}_{\mathsf{TM}}$.

An important problem with TMs is whether they loop or not. Let's imagine we have a TM that answers this question once and for all. It would be wonderful if this TM turned out to be a decider. The language is defined as:

$$\mathsf{HALT}_{\mathsf{TM}} = \big\{\langle M, w\rangle \;\big|\; M \text{ is a TM and } M \text{ halts on input } w.\big\}$$

Here is a simulator TM for this language:

```
1. Simulate $M$ on input $w$.

2. If $M$ enters its accept state
       ACCEPT.

3. If $M$ enters its reject state
       ACCEPT.
```

Unfortunately, if the original TM loops on w, this simulator loops too. So it is a recognizer, not a decider.

Imagine there is a decider Turing Machine TM03 for $\mathsf{HALT}_{\mathsf{TM}}$ that can be roughly described as follows:

```
1. Analyze $M$ and $w$ using very sophisticated techniques. Deter-
   mine whether it loops.

2. If $M$ loops on $w$
       REJECT.

3. Else
       ACCEPT.
```

In that case, we can construct a new Turing machine TM04. This new TM calls TM03 as a subroutine and gives it the input $\langle M, \langle M \rangle \rangle$. This is the same trick we used for A_{TM}.

$$\text{TM04}\big(\langle M \rangle\big) = \begin{cases} \text{HALT} & \text{if TM03 rejects } \langle M, \langle M \rangle \rangle \\ \text{LOOP} & \text{if TM03 accepts } \langle M, \langle M \rangle \rangle \end{cases}$$

Now consider TM04$\big(\langle TM04 \rangle\big)$. Once again we have a contradiction:

- If TM04$\big(\langle TM04 \rangle\big)$ halts, then it must loop.
- If TM04$\big(\langle TM04 \rangle\big)$ loops, then it must halt.

That means TM04 and therefore TM03 cannot exist. HALT_{TM} is not decidable.

An alternative way of seeing TM04 is this: Make an infinite table of all TMs and strings as usual and check halting status. TM03 is assumed to be a decider so we can fill the whole table with *"H"* or *"L"*. Then, using Cantor's diagonal argument, construct a row that is NOT on the table. This is TM04.

Closure Under Complementation: Decidable languages are closed under complementation. Can we say the same thing for recognizable languages?

Theorem: A language is decidable if and only if both the language and its complement are Turing-recognizable.

Proof: Suppose both L and $\overline{L}$ are Turing-recognizable. Any given string w is either in L or in $\overline{L}$. Therefore if we run two machines in parallel, one of them will accept and halt.

Corollary: $\overline{\text{A}_{\text{TM}}}$ and $\overline{\text{HALT}_{\text{TM}}}$ are not Turing-recognizable.

12.4 Reductions

Many problems about decidability can be reduced to other such problems. Given a TM that decides a problem, we can slightly modify it or use it as a subroutine to decide other problems just as we do with usual computer programs.

- If we know that a solution for problem A does not exist,
- If a solution for problem B can be used to obtain a solution for problem A,

Then the logical conclusion is that a solution for problem B does NOT exist.
Let's see an application of this idea on A_{TM} and HALT_{TM}.

Remember that we assumed Turing machines TM01 and TM03 decided these languages. Let M be any TM and w an input string. The actions of these TMs can be summarized as:

$M(w)$		$\text{TM01}\big(\langle M, w\rangle\big)$	$\text{TM03}\big(\langle M, w\rangle\big)$
ACCEPT		ACCEPT	ACCEPT
REJECT	$\Rightarrow$	REJECT	ACCEPT
LOOP		REJECT	REJECT

Assume TM03 exists. Then, we can construct TM01 as follows:

- Given $\big(\langle M, w\rangle\big)$, run TM03 on this input.
- If TM03 rejects, REJECT. (Because there is an infinite loop.)
- If TM03 accepts, simulate M on w.
  - If M accepts w, ACCEPT.
  - If M rejects w, REJECT.

We know TM01 does not exist. Therefore TM03 does not exist either.

This was an alternative proof for undecidability of $\mathsf{HALT_{TM}}$ using the undecidability of $\mathsf{A_{TM}}$.

We can reverse the roles of $\mathsf{A_{TM}}$ and $\mathsf{HALT_{TM}}$. Given that we have already proved TM03 does not exist, what can we say about TM01?

Assume TM01 exists. Construct TM03 as follows:

- Input $\big(\langle M, w\rangle\big)$ to TM01.
- Input $\big(\langle \overline{M}, w\rangle\big)$ to TM01.
  ($\overline{M}$ can be obtained from M by interchanging accept and reject states.)
- If one of them accepts, ACCEPT.
- If both of them reject, REJECT.

This construction gives us TM03 and it is a decider. But we know TM03 does not exist, so our assumption is wrong. TM01 does not exist.

Example 12–8: Consider $\mathsf{E_{TM}}$, the language of TMs that reject every input.

$$\mathsf{E_{TM}} = \big\{\langle M\rangle \;\big|\; L(M) = \varnothing\big\}$$

Is $\mathsf{E_{TM}}$ decidable?

Solution: No. Suppose it is decidable, and the TM A decides it. Using A, we could construct another TM B that decides $\mathsf{A_{TM}}$. The basic idea is to construct M_2 such that:

$$L(M_2) \neq \varnothing \iff M \text{ accepts } w$$

B works as follows:

```
1. Given $\langle M, w\rangle$ to $B$, construct a new TM $M_2$:

     • If input $= w$
             Run $M$ on $w$
             If it accepts
                 ACCEPT.

     • Else
             REJECT.

2. Now run $A$ with input $\langle M_2\rangle$.

     • If $A$ accepts
             $B$ REJECTS.

     • If $A$ rejects
             $B$ ACCEPTS.
```

B decides $\mathsf{A_{TM}}$, which is impossible.
Note that the TMs M and M_2 may or may not loop. But we are not running them. We are only running A which is a decider. It is guaranteed not to loop.

Example 12–9: Consider A111$_{\text{TM}}$, the language of TMs that accept the input 111. Is A111$_{\text{TM}}$ decidable?

Solution: This looks much easier. We just have to decide whether a given TM accepts the string 111 or not. But once again, the answer is no.

Suppose A111$_{\text{TM}}$ is decidable, and the TM A decides it. Using A, we could construct another TM B that decides A$_{\text{TM}}$.

The basic idea is to construct a TM N such that:

$$N \text{ accepts } 111 \iff M \text{ accepts } w$$

N works as follows:

```
• If input = $111$
      Run $M$ on $w$
      If it accepts
          ACCEPT.

• Else
      REJECT.
```

$B(\langle M, w\rangle)$ is computed as follows:

```
• Construct $N$ and run $A$ with input $\langle N\rangle$.

• If $A$ accepts $N$
      ACCEPT

• Else
      REJECT
```

B decides A$_{\text{TM}}$, which is impossible.

Mathematics and Halting: There is another way we can look at A_{TM} and HALT_{TM}. If they were decidable, we could solve lots of problems in mathematics easily!

For example, *Goldbach's Conjecture* states that all positive even integers starting with 4 can be expressed as the sum of two primes. It was first discovered in 1700s and is still not proved. We could solve it with a few lines of code if we had a decider for A_{TM}. (If TM01 existed. TM03 would also give equivalent results.)

Our program will start with $n = 4$ and check even integers one by one. It will HALT and ACCEPT as soon as it finds a number that is not the sum of two primes. (It will never REJECT.) Then, we will not run this program, but run TM01 using this input. We will know after a finite number of steps whether there is a solution or not.

Another example is *Fermat's Last Theorem*, which states that there's no integer $n \geqslant 3$ where some positive integers x, y, z satisfy

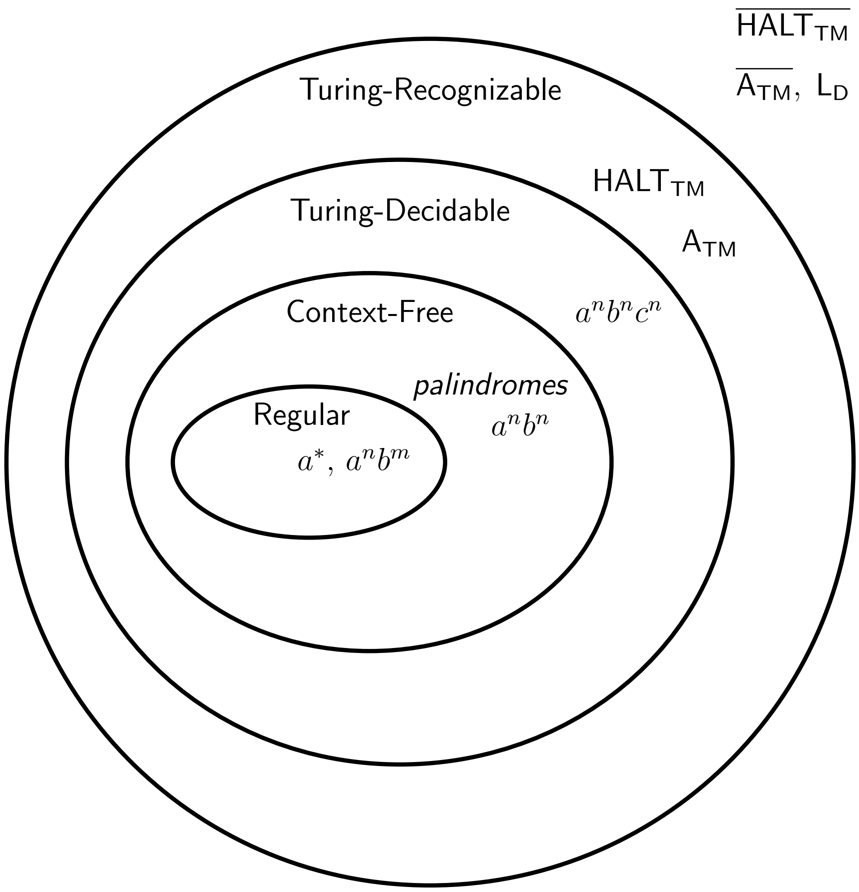
$$x^n + y^n = z^n$$

It took mathematicians roughly 350 years to prove this, but if TM01 existed, we could solve it with a very basic program any programmer can write in half an hour in any language. The idea is the same, check all possibilities one by one. We know that positive integers or sets of 4 positive integers x, y, z, n is countable. We can list them and try them one by one.

The problem is, this simple program would never halt if there were no solutions. But TM01 always halts, and it would tell us whether our program halted or looped.

If you see that *Goldbach's Conjecture*, *Fermat's Last Theorem*, and many other totally unrelated problems could all potentially be solved by TM01 or TM03, it will be clear why they cannot exist!

The language families we have seen so far, together with examples for each are given below:



Where are the finite languages?

EXERCISES

Are the following infinite sets countable or uncountable?

- If your answer is countable, describe a way of making a list of the elements in given set.
- If your answer is uncountable, give details of how to use Cantor's diagonal argument for this set.

12–1) Positive multiples of 3.

12–2) The set of all integer coefficient polynomials.

12–3) The set of all infinite strings on $\Sigma = \{0, 1, 2\}$.

12–4) The set of all words that can be generated with the alphabet $\Sigma = \{a, b, c, d, e\}$ (Words are always of finite length)

12–5) The set of all infinite strings that can be generated with the alphabet $\Sigma = \{a, b, c, d, e\}$

12–6) The set of all 2×2 matrices with entries from $\mathbb{Q}$.

12–7) The set of all 2×2 matrices with entries from $\mathbb{R}$.

12–8) The set of all words that can be written using digits $\{0, 1, \ldots, 9\}$.

12–9) Is a countable union of countable sets countable?

12–10) Is the set of all subsets of a countable set countable?

12–11) Consider the set of all numbers that are solutions to a second degree equation with integer coefficients. Is this set countable?
For example, the equation

$$x^2 - x - 1 = 0$$

has solutions:

$$x_1 = \frac{1+\sqrt{5}}{2}, \quad x_1 = \frac{1-\sqrt{5}}{2}.$$

So x_1 and x_2 are in this set.

Are the following languages decidable?

12–12) $\text{EQ}_{\text{TM}} = \{\langle M, N\rangle \mid L(M) = L(N)\}$

12–13) $\text{REG}_{\text{TM}} = \{\langle M\rangle \mid L(M) \text{ is regular.}\}$

12–14) $\text{FIN}_{\text{TM}} = \{\langle M\rangle \mid L(M) \text{ contains finitely many strings.}\}$

12–15) $\text{EVEN}_{\text{TM}} = \{\langle M\rangle \mid L(M) \text{ contains even number of strings.}\}$

12–16) $\text{ONE}_{\text{TM}} = \{\langle M\rangle \mid L(M) \text{ contains only one string } w.\}$

ANSWERS

12–1) Countable

12–2) Countable

12–3) Uncountable

12–4) Countable

12–5) Uncountable

12–6) Countable

12–7) Uncountable

12–8) Countable

12–9) Yes

12–10) No

12–11) Yes

12–12) Not Decidable

12–13) Not Decidable

12–14) Not Decidable

12–15) Not Decidable

12–16) Not Decidable

Week 13

$\mathcal{P}$ and $\mathcal{NP}$

13.1 Time Complexity

Up to now, we have studied TMs (or equivalently, algorithms) mostly in terms of their looping possibility. We were interested in whether a TM is a decider or recognizer.

Now, we will focus on deciders and study how efficient they are. In other words, given an algorithm that always solves a problem in finite time for all inputs, we are interested in finding out if the computation time is reasonably short or too long.

Time Complexity of an algorithm is a function $f(n)$ that gives the maximum number of steps the algorithm uses on any input of length n. Note that:

- The emphasis is not on time as measured by a clock. Speed of hardware, access times and such things are not taken into account. We count the number of operations.
- We take the worst time, not the best or average. If we search for a key in an unsorted list of n items, the time complexity is n. (Although we can find it in one step if we are lucky.)

Big-Oh: Given two increasing functions, how can we compare their speeds? For example, the functions $f(n) = n^2$ and $g(n) = n^3$ both have limit infinity, but we can intuitively see that $g(n) = n^3$ is in some sense faster. Actually, it is even faster than $1000n^2$ for sufficiently large n. We will formalize this intuitive notion with Big-Oh notation:

Let $f(n), g(n)$ be two functions. We say f is **Big-Oh** of g and write

$$f(n) = O\big(g(n)\big)$$

if

$$0 \leqslant f(n) \leqslant cg(n)$$

for all $n \geqslant n_0$.

For example,

$$2n^2 + 8n + 6 = O\big(n^2\big)$$

because $2n^2 + 8n + 6 < 5n^2$ for $n \geqslant 4$. (or $2n^2 + 8n + 6 < 3n^2$ for $n \geqslant 9$.) The "best" values of c and n_0 do not matter as long as we can find some that work.

Note that we can also say

$$2n^2 + 8n + 6 = O\big(n^3\big)$$

That's correct, but gives us less information about the function.

Another example is,

$$8 \log n + 5n = O\big(n\big)$$

(Choose $c = 6$ and $n_0 = 10$.)

In Big-Oh notation, adding a constant, multiplying by a constant or adding a slower increasing function does not change anything. For example, $O\big(c+n^k\big) = O\big(n^k\big)$, $O\big(c\cdot n^k\big) = O\big(n^k\big)$ and $O\big(n^k+n^\ell\big) = O\big(\,n^{\max(k,\ell)}\,\big)$.

Example 13–1: By finding appropriate c and n_0 values, show the following:

a) $(n+1)^2 = O\big(n^2\big)$

b) $n^k = O\big(n^{k+1}\big)$

c) $\log n = O\big(\ln n\big)$

d) $\ln n = O\big(\log n\big)$

e) $2^n = O\big(n!\big)$

f) $n! = O\big(n^n\big)$

g) $n^4 = O\big(2^n\big)$

h) $n^8 = O\big(2^n\big)$

Solution: a) $c = 2, \quad n_0 = 3.$

b) $c = 1, \quad n_0 = 1.$

c) $c = 1, \quad n_0 = 1.$

d) $c = \ln 10, \quad n_0 = 1.$

e) $c = 1, \quad n_0 = 4.$

f) $c = 1, \quad n_0 = 1.$

g) $c = 1, \quad n_0 = 16.$

h) $c = 1, \quad n_0 = 48.$

Note that $\log n, \ln n$ and $\log_2 n$ are equivalent in this respect.

As a summary of this topic, we can give the following ranking of functions commonly used for measuring time complexity:

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < n! < n^n$$

Example 13–2: Given a sorted array A, write an algorithm in pseudocode that determines the location of element x. Then, find the time complexity of that algorithm.

Solution: This is the binary search problem. At each step, we can reduce the search space by half:

```
INPUT Array $A$, element $x$
low $= 1$
high $=$ size($A$)
While low $\leqslant$ high
    mid $=$ (low $+$ high)$/2$
    If $A$(mid) $< x$
        low $=$ mid $+ 1$
    ElseIf $A$(mid) $> x$
        high $=$ mid $- 1$
    Else
        Return mid
    End
End
Return NOT FOUND
```

If we are lucky, $x = A(1)$ but we have to assume the worst. In that case, the complexity is $O\big(\log_2 n\big)$.

Example 13–3: The following program checks if all elements of the given array are distinct. It returns FALSE if any two are the same. What is the time complexity?

```
INPUT: Array $A$
$n = $ size$(A)$
For $i = 1$ to $n - 1$
    For $j = i + 1$ to $n$
        If $A(i) == A(j)$
            Return FALSE
        End
    End
End
Return TRUE
```

Solution: At first, we make $n - 1$ comparisons, and then we make $n - 2$ comparisons, etc.

$$(n-1) + (n-2) + (n-3) + \cdots + 2 + 1 = \frac{(n-1)n}{2}$$

So the time complexity is: $O\big(n^2\big)$

Note that, for our purposes, this algorithm is equivalent to the following:

```
For $i = 1$ to $n$
    For $j = 1$ to $n$
        $\cdots$ // One step operation
    End
End
```

They both have $O\big(n^2\big)$ steps.

Example 13–4: Write an algorithm in pseudocode that determines if the element x occurs in a given unsorted array A or not. What is the time complexity?

Solution:

```
INPUT: Array $A$, element $x$
$n =$ size($A$)
For $i = 1$ to $n$
    If $x == A(i)$
        Return TRUE
    End
End
Return FALSE
```

The complexity is $O\big(n\big)$.

Example 13–5: Write an algorithm in pseudocode that sorts a given array A. It should work by finding the smallest element and placing it as the first in each iteration. What is the time complexity?

Solution:

```
INPUT: Array $A$
$n =$ size($A$)
For $i = 1$ to $n$
    index $= i$
    For $j = i + 1$ to $n$
        If $A(j) < A($index$)$
            index $= j$
        End
    End
    Swap($A(i), A($index$)$)
End
```

The complexity is $O\big(n^2\big)$.

Example 13–6: Find the time complexity of the following algorithm based on input size n. You may assume n is a power of 2 for simplicity.

```
INPUT $n$
$S = 0,\ k = 1$
While $k \leqslant n$
    For $j = 1$ to $n$
        $S = S + 1$
    End
    $k = 2 * k$
End
```

Solution: There are n operations at each step of the while loop. We enter the while loop $\log_2 n$ times, so:

$n + n + n + \cdots + n = O\big(n \log_2 n\big)$

Example 13–7: Find the time complexity of the following algorithm based on input size n.

```
INPUT $n$
$S = 0,\ k = 1$
While $k \leqslant n$
    For $j = 1$ to $k$
        $S = S + 1$
    End
    $k = 2 * k$
End
```

Solution: $1 + 2 + 4 + 8 + \cdots + n = 2n - 1 = O\big(n\big)$

Example 13–8: We know that `Func1(n)` is an algorithm of complexity $O(n)$ and `Func2(n)` is an algorithm of complexity $O(n^2)$. Find the complexity of the following algorithm that uses these as subroutines:

```
INPUT n
For $i = 1$ to $3n$
    Func1(i)
    Func2(i)
End
For $j = 1$ to $n$
    Func1(n)
End
```

Solution: We have to add these contributions separately:

`Func1(n)` in the first loop:
$1 + 2 + \cdots + 3n = O(n^2)$

`Func2(n)` in the first loop:
$1^2 + 2^2 + \cdots + (3n)^2 = O(n^3)$

`Func1(n)` in the second loop:
$n + \cdots + n = O(n^2)$

Adding all three: $O(n^2) + O(n^3) + O(n^2) = O(n^3)$

Note that
$1^2 + 2^2 + 3^2 + \cdots + n^2 = \dfrac{n(n+1)(2n+1)}{6}$
but these details are not important for our purposes. Addition or multiplication by constants does not change the complexity class. It is sufficient to see that:
$1^2 + 2^2 + 3^2 + \cdots + n^2 = nO(n^2) = O(n^3)$

Example 13–9: We are trying to crack a password of n digits using brute force method. How many operations do we need if the characters used in the password are:

a) Decimal digits $\{0, 1, 2, \ldots, 9\}$?

b) Binary digits $\{0, 1\}$?

Solution: a) Once again, we do not need the algorithm. (It is obvious anyway.) We have to try all possibilities, and there are 10^n of them. So running time is $O\big(10^n\big)$.

b) Similar to part a), the running time is $O\big(2^n\big)$.

We call both of these cases **exponential** running times.

Example 13–10: We are given a set of n elements and we have to print the list of all the subsets. Assume printing a single subset is one operation. (Independent of number of elements of the subset.)

How many operations do we need?

Solution: This time, the algorithm is not so easy. There are different approaches. For example, we can list subsets of $0, 1, 2, \ldots, n$ elements, or for each element, we can make a decision to include or exclude.

Eventually, we will do as many operations as there are subsets, that is:

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = 2^n$$

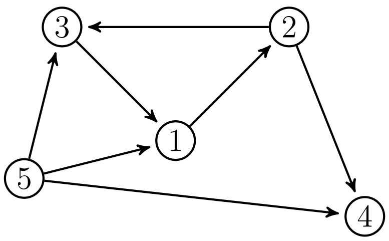
operations.

So the running time is exponential: $O\big(2^n\big)$.

Note About Graphs: Two common ways to store a graph are **adjacency matrix** and **adjacency list**.

- In an adjacency matrix, rows and columns are vertices. Matrix entries are edges.
- In an adjacency list, there is a list of neighbors associated with each vertex.

For example, the graph



can be represented as

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 1 & 0 \\ 3 & 1 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 1 & 0 & 1 & 1 & 0 \end{array}$$

or

$$\begin{array}{lcl} 1 & \to & \{2\} \\ 2 & \to & \{3,4\} \\ 3 & \to & \{1\} \\ 4 & \to & \{\} \\ 5 & \to & \{1,3,4\} \end{array}$$

We can make obvious modifications for weighted graphs, undirected graphs, etc. For a graph with n vertices, checking if there is an edge between two given vertices takes $O(1)$ operations for matrix representation and $O(n)$ operations for list representation. The difference is polynomial so it is not important for our purposes.

Example 13–11: Write an algorithm that checks whether a given undirected graph is connected. Find the time complexity of the algorithm. (Assume the graph contains n vertices and m edges.)

Solution: The idea is to check if we can reach all vertices in the graph starting with any vertex.

```
INPUT Undirected Graph $G$
Create an empty stack $S$
Choose any vertex.  Mark it and push to $S$
While $S$ is not empty
    Pop vertex $v$
    For all unmarked neighbors $u$ of $v$
        Mark $u$ and push $u$ to $S$
    End
End
For all vertices $w \in G$
    If $w$ is not marked
        Return FALSE
    End
End
Return TRUE
```

Here we use DFS (Depth First Search). Choosing a queue instead of a stack would be equivalent to using BFS (Breadth First Search).

We iterate the while loop at most n times.

- Using adjacency matrix, each iteration takes n operations. Therefore complexity is $O\big(n^2\big)$.
- Using adjacency list, we do as many operations as there are edges. Therefore complexity is $O\big(n+m\big)$.

Example 13–12: You are given a directed graph with n vertices and m edges. Write an algorithm that checks whether it contains a cycle.

Solution: The following recursive algorithm uses Depth First Search to detect a cycle. It returns TRUE if there is a cycle.

```
INPUT Directed Graph $G$
Color all vertices white
For each vertex $v \in G$
    If $v$ is white
        DFS$(G, v)$
    End
End
Return FALSE

DFS$(G, u)$
Color $u$ gray
For all its children $w$
    If $w$ is gray
        Return TRUE
    ElseIf $w$ is white
        DFS$(G, w)$
    End
End
Color $u$ black
```

It is more difficult to find complexity. But once again we find $O\big(n^2\big)$ for adjacency matrix and $O\big(n+m\big)$ for adjacency list representations.

13.2 The Class $\mathcal{P}$

Consider a TM for the language 0^n1^n. If it is a single tape, deterministic TM and if it checks strings by crossing off one 0 and then one 1, it needs n iterations and will make $O(n)$ moves at each iteration. That means the algorithm is $O(n^2)$.

If we use a two-tape TM and replicate the input $O(n)$ steps, we can finish the rest in $O(n)$ steps, so this algorithm is $O(n)$.

As you can see, running times are related to the computational model. But, we consider polynomial differences in running times unimportant. (Although they are *very* important in practice.)

We want to think of the above two algorithms as essentially the same, so we define $\mathcal{P}$ as the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine.

In other words,

$$\mathcal{P} = \bigcup_{k=0}^{\infty} \mathsf{TIME}(n^k)$$

The class $\mathcal{P}$ is very important because it roughly corresponds to the algorithms that give results in reasonable times, in other words, all useful computer programs.

If we compare polynomial and exponential time algorithms, the difference is striking. Suppose an algorithm gives an answer in one hour for input size $n = 50$. Then we make a second run for $n = 100$.

If it is a polynomial time algorithm with complexity $O(n)$, the second run will be completed in two hours. If it is $O(n^2)$ it will be four hours, if it is $O(n^3)$ it will be eight hours, etc.

But if the algorithm is exponential, the second run will take
$2^{50} = 1$ quadrillion hours $= 128$ billion years!

Example 13–13: You are given a set of n positive integers. You have to find two distinct integers p, q from this set such that $p+q$ is as close as possible to 256.

Show that this problem is in $\mathcal{P}$.

Solution: This is clearly $O(n^2)$. We can even see this without any code because there are $\dfrac{n(n-1)}{2}$ pairs to check.

```
INPUT Integer array $A$, array size $n$.
$p = A(1), \quad q = A(2)$
For $i = 1$ to $n-1$
    For $j = i+1$ to $n$
        If $|A(i) + A(j) - 256| < |p + q - 256|$
            $p = A(i), \quad q = A(j)$
        End
    End
End
Return $p$, $q$
```

Example 13–14: There are n students in a class. We want to find 3 students A, B, C such that the couples $A-B, A-C$ and $B-C$ have been lab or project partners in some course. Each student gives us a list of all students in the class he/she ever worked with.

Show that this problem is in $\mathcal{P}$.

Solution: There are $\binom{n}{3} = O(n^3)$ triples. We can check each one in $O(n)$ operations by going over lists. That makes $O(n^4)$ operations. Therefore the problem is in $\mathcal{P}$.

Example 13–15: (String Matching Problem)
You are given a string of length n. You are also given a word (a second string) of length k, where $k \leqslant n$. You want to determine if the word occurs in the string.

Show that this problem is in $\mathcal{P}$.

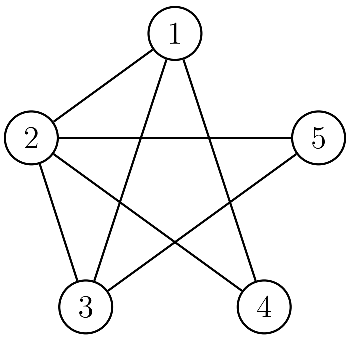
Solution: Any string element between 1 and $n-k+1$ is a possibility for the start of a match. So we try each. If there's a mismatch, we give that one up and continue with the next possible position.

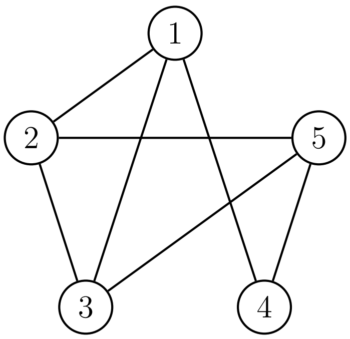
```
INPUT String $A[n]$, Word $B[k]$
For $i = 1$ to $n - k + 1$
    test = TRUE
    For $j = 1$ to $k$
        If $A[i + j - 1] \neq B[j]$
            test = FALSE
            Break
        End
    End
    If test == TRUE
        Return TRUE
    End
End
Return FALSE
```

This algorithm uses nk steps therefore it is $O\big(n^2\big)$.

So the problem is in $\mathcal{P}$.

Example 13–16: (Euler Path Problem)
An Euler path in a graph G is a path that goes through each edge once. For example, consider the following graphs:





A solution for the left graph is $1-3-5-2-4-1-2-3$. There are other solutions. Note that all such paths either start with 1 and end with 3 or vice versa.

There is no solution for the right one.

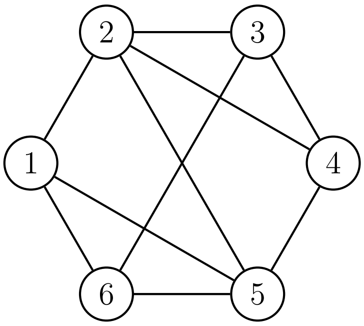
Is Euler path problem in $\mathcal{P}$?

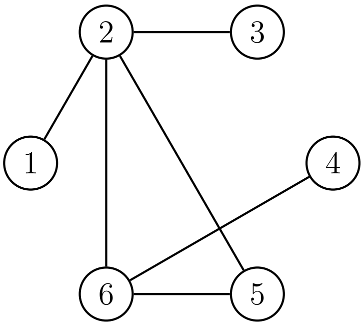
Solution: We can easily prove that a graph G contains an Euler path from vertex s to a different vertex t if and only if:

- G is connected,
- s and t have odd degree,
- all other vertices of G have even degree.

These conditions can be checked in polynomial time so the problem is in $\mathcal{P}$.

Example 13–17: (Hamiltonian Path Problem)
A Hamiltonian path in a graph G is a path that goes through each vertex once.
For example, consider the following graphs:





A solution for the left graph is: $1 - 2 - 3 - 6 - 5 - 4$. There is no solution for the right one.

Is Hamiltonian path problem in $\mathcal{P}$?

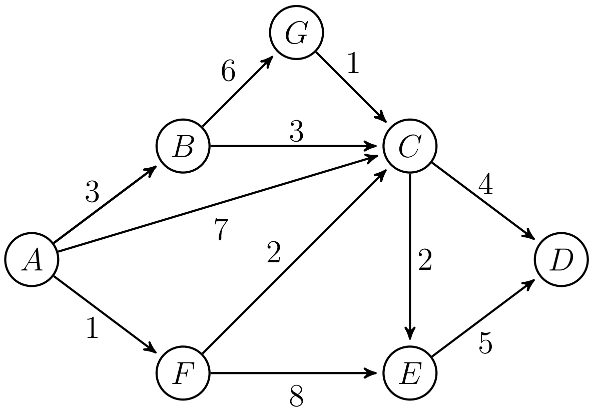
We don't know! We can try all possible arrangements of vertices and check if there is such a path. This is a brute-force approach.

$$n \text{ vertices} \quad \Rightarrow \quad n! \text{ possible orderings}$$

This is clearly not a polynomial. Actually, it is even worse than exponential.

Example 13–18: (Shortest Path Problem)
We consider a directed graph with nonnegative weights:

Given two vertices, what is the path of minimum total weight connecting them?



First of all, this is an optimization problem, but $\mathcal{P}$ contains decision problems only. We have to modify the question such that the answer is *Yes* or *No*. For example, we can ask:

Is there is a path of weight less than W connecting given vertices?

Is the Shortest Path Problem in $\mathcal{P}$?

Solution: A solution is Dijkstra's algorithm where we start from the source vertex and find the shortest distance to all vertices that can be reached in 1 step, then 2 steps, etc.

For example, the shortest path in the graph above from A to D can be found as $A - F - C - D$ and its weight is 7.

For a graph with n vertices and adjacency matrix representation, the complexity will be $O(n^2)$. So the answer is yes.

13.3 The Class $\mathcal{NP}$

The term $\mathcal{NP}$ means nondeterministic polynomial time algorithm.

We can characterize $\mathcal{NP}$ in two different, but equivalent ways:

- $\mathcal{NP}$ is the class of languages that have polynomial time verifiers.
- $\mathcal{NP}$ is the class of languages decided by some nondeterministic polynomial time Turing machine.

Here, the basic idea is that verifying (in other words, checking the correctness of) a solution is much easier than finding the solution. For example, if we were given a Hamiltonian path, we could check its correctness in polynomial time.

We can show equivalence as follows:

- If we have polynomial time verifier, we can use it at every branch of the NTM computation tree. That's like the same program running at a very large number of parallel computers, each starting with a different input and covering all possibilities.
- If we have a NTM, one of the branches finds a solution in polynomial time. Then we can use the choices made by that machine to build a verifier.

Example 13–19: Coloring Problem

Given a graph with n vertices and an integer k, is there a way to color the vertices with k colors such that adjacent vertices are colored differently?

Is this problem in $\mathcal{NP}$?

Solution: We are NOT trying to solve the Coloring Problem. We assume we are given a solution for a specific graph. The question is, can we verify it in polynomial time?

There are $\dfrac{n(n-1)}{2} = O\big(n^2\big)$ pairs to check, so the answer is yes:

```
For $i = 1$ to $n - 1$
    For $j = i + 1$ to $n$
        If vertices $i - j$ are connected
            If Color($i$) == Color($j$)
                Return REJECT
            End
        End
    End
End
Return ACCEPT
```

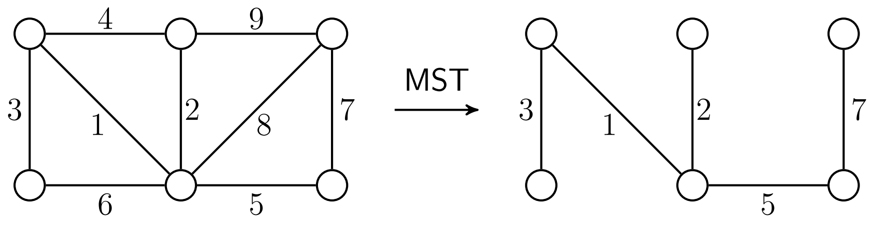
This algorithm takes $O\big(n^2\big)$ time, i.e. polynomial time.

Therefore Coloring Problem is in $\mathcal{NP}$.

Note that we assume an adjacency matrix is used for the graph. If an adjacency list is used the complexity will be $O\big(n^3\big)$, but such polynomial differences are not important. The problem is still $\mathcal{NP}$.

Example 13–20: Minimum Spanning Tree Problem

A minimum spanning tree for a weighted, connected, undirected graph G is a subgraph that connects all vertices and has minimum weight. For example,



Once again, we have to express this optimization problem as a decision problem:

Given a weighted graph G and a number W, is there a spanning tree with weight $\leqslant W$?

Is the MST problem in $\mathcal{NP}$?

Solution: We can solve the MST problem using Kruskal's algorithm where we add edges one by one starting with the minimum weight edge. (We continue adding as long as there is no cycle.) But we are NOT trying to solve MST!

Assume a solution is given for a specific graph G and number W. If there are n vertices in G, the solution contains $n-1$ edges.

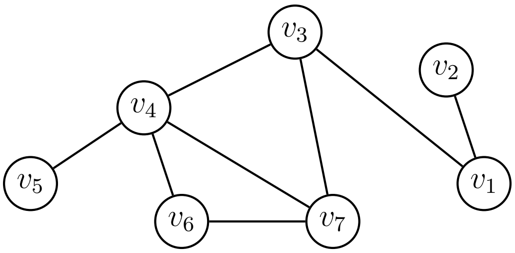
We can check each edge in $O(1)$ (or $O(n)$) steps. Then we will add edge weights and compare to W. We also have to check that the resulting graph is connected. (Or we should check if there is a cycle. These conditions are equivalent.)

All of this is polynomial, so the problem is in $\mathcal{NP}$.

Example 13–21: Vertex Cover Problem

Given a graph and an integer k, is there a collection of k vertices such that each edge is connected to one of the vertices in the collection?

For example, for the following graph, a $3-$cover exists: v_1, v_4, v_7. But there is no $2-$cover.



Is this problem in $\mathcal{NP}$?

Solution:
```
INPUT  A graph $G$ with $n$ vertices, $m$ edges.
       A list of $k$ vertices.
For $i = 1$ to $k$
    Mark all edges of $v_i$
End
For $i = 1$ to $m$
    If $E_i$ is unmarked
        Return REJECT
    End
End
Return ACCEPT
```

The complexity of the verifying algorithm is $O(n^2)$.

Therefore Vertex Cover Problem is in $\mathcal{NP}$.

EXERCISES

Find the time complexity of the following algorithms:

13–1)

```
INPUT $n$
$S = 0$
For $i = 1$ to $n$
    For $j = 1$ to $i^2$
        $S = S + j$
    End
End
```

13–2)

```
INPUT $n$
Create $n \times n$ empty
    matrix $A$.
For $i = 1$ to $n$
    For $j = 1$ to $n$
        $A(i, j) = i^2 + j^2$
    End
End
For $i = 1$ to $n$
    If $i$ is even
        $A(i, i) = 1$
    Else
        $A(i, i) = -1$
    End
End
```

13–3)

```
INPUT $n$
$S = 0$
For $i = 1$ to $n$
    For $j = i$ to $2 * n$
        For $k = 1$ to $20$
            $S = S + 1$
        End
    End
End
```

13–4)

```
INPUT $n$
$S = 0$
For $i = 1$ to $n$
    If $i$ is even
        For $j = 1$ to $n^2$
            $S = S + 1$
        End
    Else
        $S = S + 10$
    End
End
```

13–5) You are given two $n \times n$ matrices. You want to multiply them using the standard textbook method. What is the time complexity?

13–6) You are given an $n \times n$ matrix. You want to find its determinant using cofactor expansion. What is the time complexity?

Write an algorithm in pseudocode for the following decision problems and find the time complexity of those algorithms. Are the problems in $\mathcal{P}$?

13–7) Language of palindromes in $\Sigma = \{a, b\}$.

13–8) You are given n distinct positive integers where $n \geqslant 3$. You want to find the third largest integer.

13–9) You are given an undirected graph with n vertices. Does it contain a rectangle?

In other words, does it contain vertices A, B, C, D such that edges $A - B, B - C, C - D$ and $D - A$ are in the graph?

13–10) You are given a set of n distinct positive integers. Are there two integers p, q in the set such that $p = q^2$?

Show that the following problems are in $\mathcal{NP}$: (Are they also in $\mathcal{P}$?)

13–11) (Subset-Sum-Zero Problem)

We are given a set of integers $S = \{i_1, i_2, \ldots, i_n\}$. Is there a subset that adds up to 0?

For example,

$S = \{9, 18, -75, 3, 1, -40\}$	$S = \{-12, 28, 7, -66, 5\}$
No	Yes

13–12) (Bin Packing Problem)

There are n objects. Each object has size $s_i,\ i = 1, 2, \ldots, n$.

There are k containers. The size of each container is C.

We accept if the objects fit in containers and reject otherwise.

13–13) (Longest Path Problem)

Consider an undirected, weighted graph. Given two vertices, what is the weight of the longest simple path connecting them? A simple path is a path where all vertices are distinct. (First, express this as a decision problem.)

13–14) ($0-1$ Knapsack Problem)

There are n items. Each has a weight w_i and value v_i. We choose some of them and put in a knapsack of weight capacity W. What is the maximum total value we can obtain? (First, express this as a decision problem.)

Show that the following problems are in $\mathcal{NP}$: (Are they also in $\mathcal{P}$?)

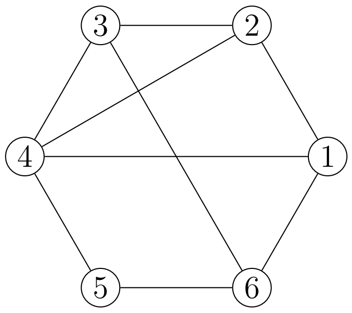
13–15) (Traveling Salesman Problem) There are n cities. A salesman wants to visit all of them. He will start from one and eventually return to the same city. All the distances are known.

Is there a path whose total length is less than a given number L?

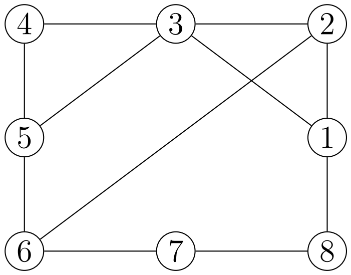
13–16) There are n courses. Each course is a vertex. If there exists one or more students registered to both courses, these vertices are connected by an edge. Finals last 10 days therefore we want to label each vertex with a number $k \in \{1, 2, \ldots, 10\}$ such that no connected vertices have the same number. (No student should take two or more exams in one day.)

Is such a labeling possible?

13–17) ($\mathbf{n - 2n}$ Clique Problem) A clique in an undirected graph is a subgraph wherein every two nodes are connected by an edge. We are given an undirected graph with $2n$ nodes. Does it have a clique with n nodes?
For example:



Yes: $1, 2, 4$



No

ANSWERS

13–1) $O(n^3)$

13–2) $O(n^2)$

13–3) $O(n^2)$

13–4) $O(n^3)$

13–5) There are $n \times n = n^2$ entries in the output, and we have to do $O(n)$ operations for each entry. That will be $O(n^3)$ operations. There will be three `for` loops.

13–6) Using cofactor expansion, we reduce the problem of size n to n problems of size $n-1$ and continue recursively until 1. That will be $O(n!)$ operations. (There are better, polynomial algorithms for finding determinant.)

13–7) We can check the given string in $O(n^2)$ operations $\quad \Rightarrow$ the language is in $\mathcal{P}$.

13–8) We can make 3 passes through the list, which will give us $3n = O(n)$ operations $\quad \Rightarrow \quad$ it is in $\mathcal{P}$.

13–9) We need to check $\displaystyle \binom{n}{4} = \frac{n(n-1)(n-2)(n-3)}{4!}$ sets of 4 vertices. This is $O(n^4)$ operations $\quad \Rightarrow \quad$ it is in $\mathcal{P}$.

13–10) We can decide the problem after checking all possible pairs. That will be $O(n^2)$ operations $\quad \Rightarrow \quad$ it is in $\mathcal{P}$.

13–11) Yes, because given a solution, we can verify it in $O(n)$ steps. (Actually, in less than n steps.)

13–12) Given a solution, we have n objects and k containers, together with the list of objects in each container. To check the solution, we have to do $n - k$ additions and k comparisons. This is $O(n)$ operations.

13–13) The decision version is:
Is there is a path of weight more than W connecting given vertices?
This is in $\mathcal{NP}$. Given a graph with n vertices and a specific solution, the path length is at most $n - 1$. Checking this takes polynomial time.

13–14) The decision version is:
Is there is a subset of items with weight $\leqslant W$ and value $\geqslant V$?
This is in $\mathcal{NP}$. A specific solution contains k items where $k \leqslant n$ therefore we can check it in $O(n)$ operations.

13–15) Given a solution (a sequence of vertices), we will look up the weights of the edges connecting them and then add them all and compare with L. This can be done in polynomial time. $(O(n))$

13–16) Given a solution, we have to check each course (vertex). To check one vertex, we have to check all its neighbors. That makes a total of $O(n^2)$ operations. This is polynomial time.

13–17) Yes, we can verify the solution in polynomial time. There are $\frac{n(n-1)}{2} = O(n^2)$ pairs of vertices in the clique. We can check them in polynomial time.

Week 14

$\mathcal{NP}$–Complete Problems

14.1 $\mathcal{P}$ versus $\mathcal{NP}$

We can summarize the previous chapter as follows:

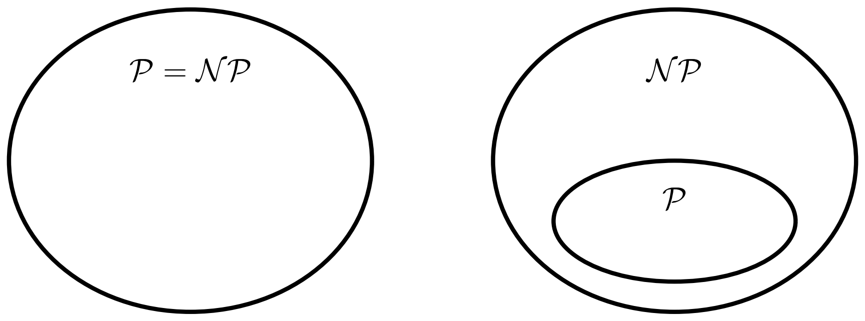
- $\mathcal{P}$ is the class of problems with polynomial time solutions.
- $\mathcal{NP}$ is the class of problems with polynomial time verifiers.

Clearly, any problem in $\mathcal{P}$ is also in $\mathcal{NP}$. But are there any problems in $\mathcal{NP}$ and not in $\mathcal{P}$?

We do NOT know. There are many problems in $\mathcal{NP}$ (like Hamiltonian Path Problem) for which we do not have any solution algorithms that are polynomial time. But this doesn't mean some such solution cannot be found in the future.

The question of whether $\mathcal{P} = \mathcal{NP}$ is considered by many researchers as the most important open problem in computer science.

Many people working in computational complexity became aware of the problem in 1950s and 1960s. Its precise formulation was done by Stephen Cook in 1971.



Are $\mathcal{P}$ and $\mathcal{NP}$ identical? Or is P a proper subset of NP? This is still unsolved.

If $\mathcal{P} = \mathcal{NP}$, this means that problems like Graph Coloring, Vertex Cover, Hamiltonian Path, Traveling Salesman, Subset–Sum, etc. have solution algorithms with polynomial time complexity. Lots of researchers have worked hard on these problems for decades but could not find a solution for the general case. (Note that many polynomial algorithms that give approximate solutions, or give the exact solution for special cases are known.)

If $\mathcal{P} \subset \mathcal{NP}$ and $\mathcal{P} \neq \mathcal{NP}$, this means those problems cannot be solved in polynomial time, however hard we try. Many experts think this is the case but it is very difficult to prove that.

Here we focus on mainly polynomial and exponential time algorithms. But there are many more complexity classes. For example,

- If an algorithm is not polynomial, it does not have to be exponential. It may belong to QP (quasi-polynomial) class. These require more steps than a polynomial but less than exponential algorithms.
- Exponential time algorithms are not the slowest. There are complexity classes beyond $\mathcal{NP}$.

14.2 Polynomial Time Reductions

Sometimes a problem can be transformed into another problem. If we already know the solution to one, we can find the solution to the other.
Problem Y is **polynomial-time reducible** to problem X if it can be solved using:

- Polynomial number of computational steps and
- Polynomial number of calls to solution method of problem X.

In that case, we use the notation

$$Y \leqslant_P X$$

Note that Y is the simpler problem. Here, the way we use the word *reduce* is different from what we usually do in mathematics. We are reducing the *easy* problem to the *difficult* one. We assume we already have a solution algorithm for the difficult one and adapt it to the easier problem.

In terms of languages, we can rephrase the definition as:

Language A is said to be polynomial-time reducible to language B if a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ exists, where, for every string $s \in \Sigma^*$

$$s \in A \iff f(s) \in B$$

and f can be computed in polynomial time.

Theorem: Let A and B be languages and $A \leqslant_P B$. Then:

$$B \in P \quad \Rightarrow \quad A \in P$$

$$A \notin P \quad \Rightarrow \quad B \notin P$$

Here are some examples of polynomial-time reductions from one algorithm to another. In each case check that the reduction can be done in polynomial time and:

$$\text{Problem } A \ \leqslant_P \ \text{Problem } B$$

1. A: *Given a set of numbers, find the second largest one.*
   B: *Given a set of numbers, sort them.*

2. A: *Given a set of integers, find if there is any duplicate.*
   B: *Given a set of integers, find the minimum distance between any two of them.*
   To reduce A to B, find minimum distance. If it is 0, there is a duplicate.

3. A: *Given two integers n, m, find gcd(n, m).*
   B: *Given two integers n, m, find lcm(n, m).*
   Using the mathematical relation

   $$\text{gcd}(n, m) \cdot \text{lcm}(n, m) = n \cdot m$$

   we can easily find the solution of one of these problems if we know the other one. In other words, both $A \leqslant_P B$ and $B \leqslant_P A$ are correct.

4. A: *Language of binary strings that start and end with 1.*
   B: *Language of binary strings that start and end with the same symbol.*

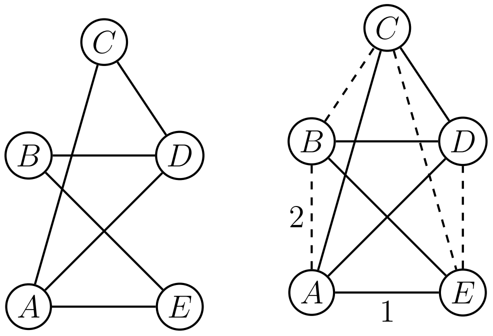
5. A: *Language of binary strings that are palindromes.*
   B: *Language of binary strings that start and end with the same symbol.*
   Here, B is the simpler problem but still $A \leqslant_P B$. To reduce A to B, we have to call B many times. What is important is that, the number of calls is $O(n)$, in other words, polynomial.

Example 14–1: Show that Hamiltonian Cycle Problem is polynomial-time reducible to Traveling Salesman Problem.

Solution: Hamiltonian Cycle Problem (HCP) is similar to Hamiltonian Path Problem. *Given an undirected graph G, is there a cycle that visits every vertex once?*

We consider the decision version of the Traveling Salesman Problem (TSP): *Given a weighted graph G and a positive number k, is there a cycle that visits each vertex once and whose total weight is less than or equal to k?*

Suppose we have an algorithm that can solve any TSP. Given a specific HCP, transform into TSP as follows: If two vertices are connected, set their weight as 1. Otherwise set the weight as 2. Set $k = n$. For example:



$ACDBEA$ cycle weight$= 5$

If we can find a path of length $\leqslant n$ it is the one we are looking for. Clearly, this reduction is of polynomial in the number of vertices n, therefore

$$\text{HCP} \leqslant_P \text{TSP}$$

Example 14–2: Consider Subset–Sum and Exact–Cover problems as defined below:

Subset–Sum Problem: We are given a set S of integers and a target number T. Is there a subset of S that adds up to T?
For example,

$S = \{-20, -5, 0, 4, 7, 19\}, \quad T = 14 \quad \Rightarrow$ Yes.

$S = \{-3, 8, 17, 25, 66\}, \quad T = 60 \quad \Rightarrow$ No.

Exact–Cover Problem: We are given a set E and a set P of subsets of E. Is there a subset of P that is an exact cover of E?

In other words, is it possible to find a collection of subsets such that each element of E is in exactly one of them?
For example, let $E = \{1, 2, 3, 4, 5\}$:

- $P = \big\{\{2\}, \{1, 2\}, \{2, 3\}, \{3, 4, 5\}, \{1, 2, 3, 4\}\big\}$
  $\Rightarrow$ Yes: $\big\{\{1, 2\}, \{3, 4, 5\}\big\}$

- $P = \big\{\{2\}, \{3\}, \{1, 2\}, \{2, 3\}, \{3, 4\}, \{1, 3, 5\}\big\} \quad \Rightarrow$ No.

Show that:

$$\text{Exact–Cover Problem} \ \leqslant_P \ \text{Subset–Sum Problem}$$

Solution: We assume we already have an algorithm that solves Subset–Sum Problem. The question is, given an Exact–Cover Problem, how can we express it as a Subset–Sum Problem?

Obviously, using the given set and subsets, we have to produce some numbers and a target number.

Here is an idea: Map each subset to the number of elements in the subset. Map the number of elements in E to target number T. For example, for

$$E = \{1,2,3,4\},\ P = \big\{\{2\},\{1,3\},\{1,4\},\{1,3,4\}\big\}$$

we have $S = \{1,2,3\}$ and $T = 4$. (There were two 2's but we eliminated one.) In this example, there is a solution to the Subset–Sum Problem, which is $1+3=4$ and the corresponding solution to Exact–Cover Problem is:

$$\{2\} \cup \{1,3,4\} = \{1,2,3,4\}$$

Still, this idea does not work in general. (Can you give counter-examples?)

A better idea is to replace each number $i \in E$ with 2^{i-1}. In this way, we eliminate the possibility of obtaining the same representative number for different subsets.

Then, we map each subset to the sum of elements of that subset and then sum of E will be T. For example, suppose
$E = \{1,2,3,4,5\},\ P = \big\{\{2\},\{1,2\},\{2,3\},\{3,4,5\}\big\}$.
Map those as:

$$1 \to 1,\quad 2 \to 2,\quad 3 \to 4,\quad 4 \to 8,\quad 5 \to 16$$

$$\{2\} \to 2,\quad \{1,2\} \to 3,\quad \{2,3\} \to 6,\quad \{3,4,5\} \to 28$$

Subset–Sum Problem is: $S = \{2,3,6,28\},\ T = 31$.

A solution is $3+28=31$ and the corresponding solution to Exact–Cover Problem is:

$$\{1,2\} \cup \{3,4,5\} = \{1,2,3,4,5\}$$

Example 14–3: Consider Set–Partition Problem:

Set–Partition Problem: We are given a multiset S of positive integers. Is it possible to find two disjoint subsets of S such that $S_1 \cap S_2 = \varnothing, \quad S_1 \cup S_2 = S$ and the sum of elements of S_1, S_2 are equal?

In a multiset, elements can be repeated. For example, let $S = \{2, 3, 3, 5, 7\}$. Then, $S_1 = \{2, 3, 5\}$, $S_2 = \{3, 7\}$ is a solution.

Show that: Subset–Sum Problem $\leqslant_P$ Set–Partition Problem

Solution: Given a Subset–Sum Problem with set S, the sum of elements M and target number T, three cases are trivial:
If $M < T$ there is no solution.
If $M = T$ the set S itself is a solution.
If $M = 2T$, use Set–Partition. Any of the subsets is a solution.

- If $T < M < 2T$ add the integer $n = 2T - M$ to S. Use Set–Partition. A solution is the subset not containing n. For example,

  $S = \{1, 3, 4, 7\}, \quad T = 12, \quad M = 15.$
  Add 9 to S, use Set–Partition: $S_1 = \{1, 4, 7\}$
  $S_2 = \{3, 9\}$, the solution is $\{1, 4, 7\}$.

- If $2T < M$ add the integer $n = M - 2T$ to S. Use Set–Partition. A solution is the subset containing n, after we remove n. For example,

  $S = \{2, 5, 8, 11, 13\}, \quad T = 18, \quad M = 39.$
  Add 3 to S, use Set–Partition: $S_1 = \{8, 13\}$,
  $S_2 = \{2, 3, 5, 11\}$, the solution is $\{2, 5, 11\}$.

14.3 $\mathcal{NP}$–Complete Problems

There are certain problems in $\mathcal{NP}$ whose complexity is related to all the problems in $\mathcal{NP}$. Before we give a full definition, let's see some examples.

Example 14–4: SAT Problem
This problem is about the satisfiability of a logical formula. We will simplify the problem by considering formulas in **Conjunctive Normal Form** only.

This form has two levels. In the first level, we connect the variables (or their negations) by OR. In the second, we connect the first level structures by AND. The following are in conjunctive normal form:

$$\big(x_1 \vee x_2 \vee x_4 \vee x_5\big) \wedge \big(\neg x_1 \vee \neg x_3\big) \wedge \big(\neg x_2 \vee x_3 \vee \neg x_6\big)$$

$$x_1 \wedge \big(\neg x_2 \vee \neg x_1\big) \wedge \big(x_3 \vee x_2\big) \wedge \big(\neg x_3 \vee x_2\big)$$

The question is: *Is it possible to find truth values 0 or 1 for each x_i such that the formula is satisfied?*

For the first example, the answer is Yes, for the second, it is No.

Show that SAT Problem is in $\mathcal{NP}$.

Solution: Given a formula of size n and a possible solution such as $x_1 = 1, x_2 = 0, \ldots$, etc. we can verify the solution in $O(n)$ operations, therefore the problem is in $\mathcal{NP}$.

Clearly, we can solve this problem by trial and error. But if input size $=$ number of variables $= n$, there will be 2^n possibilities, so the algorithm which systematically checks all cases will not be polynomial-time. We do not know if SAT is in $\mathcal{P}$ or not.

Example 14–5: 3–SAT Problem

3–SAT is a special case of the SAT problem where each first level component has exactly three variables. For example, the following is a 3–SAT formula:

$$\big(x_1 \vee \neg x_2 \vee x_3\big) \wedge \big(\neg x_1 \vee \neg x_3 \vee x_4\big)$$

But the formulas on the previous page are not.

Show that SAT problem is reducible to 3–SAT problem.

Solution: Given an SAT problem, we have to add new variables $y_1, y_2, \ldots$ and modify parentheses:

1. If it contains a single variable, say, x_1, replace it by:

$$\big(x_1 \vee y_1 \vee y_2\big) \wedge \big(x_1 \vee \neg y_1 \vee y_2\big) \wedge \big(x_1 \vee y_1 \vee \neg y_2\big) \wedge \big(x_1 \vee \neg y_1 \vee \neg y_2\big)$$

2. If it contains two variables, say, $x_1 \vee \neg x_2$, replace it by:

$$\big(x_1 \vee \neg x_2 \vee y_1\big) \wedge \big(x_1 \vee \neg x_2 \vee \neg y_1\big)$$

3. If it contains three variables, there's no need to modify.

4. If it contains n variables where $n > 3$, introduce $n - 3$ new variables and replace the parenthesis by $n - 2$ new ones:

$$\big(x_1 \vee x_2 \vee y_1\big) \wedge \big(\neg y_1 \vee x_3 \vee y_2\big) \wedge \big(\neg y_2 \vee x_4 \vee y_3\big)$$
$$\wedge \cdots \wedge \big(\neg y_{n-2} \vee x_{n-2} \vee y_{n-3}\big) \wedge \big(\neg y_{n-3} \vee x_{n-1} \vee x_n\big)$$

Suppose we have the $4-$variable formula:

$$a \vee \neg b \vee \neg c \vee d$$

We have to add new variable z and replace the above formula by:

$$\big(a \vee \neg b \vee z\big) \wedge \big(\neg z \vee c \vee d\big)$$

If there are $5-$variables:

$$a \vee b \vee c \vee \neg d \vee \neg e$$

add new variables $y,\ z$ and replace it by:

$$\big(a \vee b \vee y\big) \wedge \big(\neg y \vee c \vee z\big) \wedge \big(\neg z \vee \neg d \vee \neg e\big)$$

In each of these cases, please check that the modified formula can be satisfied if and only if the original formula can.

(Note that we are assuming variables are not repeated in a parenthesis. In other words, we assume $\big(x_1 \vee x_1 \vee x_2\big)$ is actually $\big(x_1 \vee x_2\big)$ and $\big(x_1 \vee \neg x_1 \vee x_2 \vee x_3\big)$ is redundant because it is always true.)

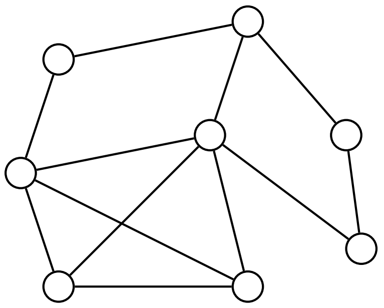
We have increased the size of the input, but if there are n variables, we are replacing each parenthesis with at most $O\big(n\big)$ new ones. Also, we are introducing at most $O\big(n\big)$ new variables per parenthesis. So the reduction

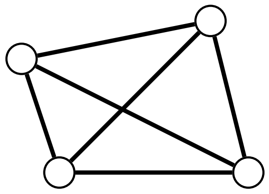
$$\text{SAT Problem} \quad \leqslant_P \quad 3\text{–SAT Problem}$$

is polynomial.

Example 14–6: k–Clique Problem

Given an undirected graph, a k–clique is a subset of k vertices such that each one is connected to all others in the clique.





A 4–clique

Clearly, if a graph has a k–clique then it has $2, 3, \ldots, k-1$ cliques. The k–clique problem is to determine whether such a clique exists for a given graph G and integer k. (Is there a polynomial time solution?)

Show that 3–SAT problem is polynomially reducible to k–clique problem.

Solution: On the surface, these two problems look very different. One is about Boolean formulas, the other about graphs. How can we find a correspondence between these concepts?

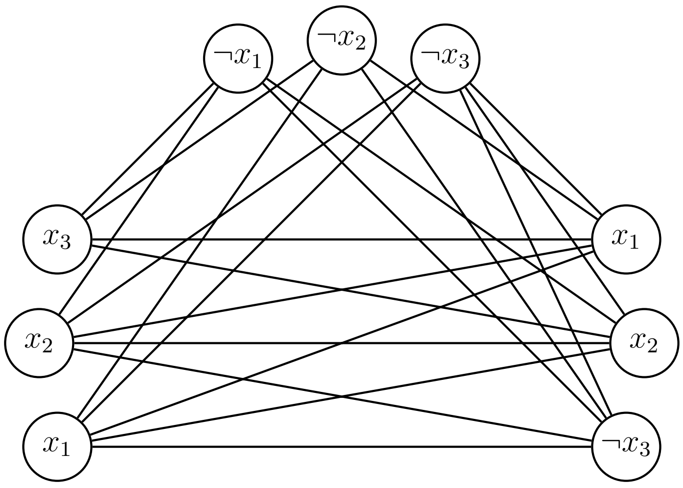
Given a 3–SAT problem containing n parentheses, we are trying to choose one variable from each parenthesis and set it to `TRUE`. This can be modeled by an n–clique in a graph of $3n$ vertices. Therefore these 3 variables must be unconnected.

Another important point is that, if x_1 is TRUE then $\neg x_1$ is FALSE. We cannot choose both, so they should also be disconnected.

But if we choose x_1 as TRUE, we can choose either x_2 or $\neg x_2$ as TRUE. These variables must be connected. They could be part of the same clique.

An example based on these ideas is:

$$\big(x_1 \vee x_2 \vee x_3\big) \wedge \big(\neg x_1 \vee \neg x_2 \vee \neg x_3\big) \wedge \big(x_1 \vee x_2 \vee \neg x_3\big)$$



Any 3–clique from this figure corresponds to a choice of variables that satisfies the Boolean formula.

For a graph constructed in that manner from the given 3–SAT problem, a k–clique exists if and only if a solution that satisfies the logical formula exists, in other words:

$$\text{3–SAT Problem} \quad \leqslant_P \quad k\text{–Clique Problem}$$

Formal Definition of $\mathcal{NP}$–Complete Problems: A language A is $\mathcal{NP}$–complete if:

- A is in $\mathcal{NP}$, and
- For every language B in $\mathcal{NP}$: $B \leqslant_P A$.

There are certain problems in $\mathcal{NP}$ whose complexity is related to all the problems in $\mathcal{NP}$. If we can solve one of these problems in polynomial time, then we can solve all problems in $\mathcal{NP}$ in polynomial time, because they are all reducible.

Cook - Levin Theorem: SAT problem is $\mathcal{NP}$–Complete.

This was proven by Stephen Cook in 1971. [1] He later received a Turing–award for his work.

We won't go into the details of the proof which is quite technical, but the main idea is clear: If a problem is in $\mathcal{NP}$, there must be a nondeterministic TM deciding it in polynomial time. Let's write all operations of the accepting branch of this TM as logical steps. We will obtain a Boolean formula that is satisfied if and only if the TM accepts and halts.

We know that SAT $\leqslant_P 3$–SAT and 3–SAT $\leqslant_P k$–Clique , so both these problems are $\mathcal{NP}$–Complete. Note that if L is $\mathcal{NP}$–Complete and we want to prove that K is also $\mathcal{NP}$–Complete, then we have to show $L \leqslant_P K$.

Theorem: If A is $\mathcal{NP}$–complete and $A \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

If we can find a polynomial-time solution to a single $\mathcal{NP}$–Complete problem, for example, k–Clique, that would constitute proof that $\mathcal{P}$ and $\mathcal{NP}$ are identical because using reductions, we would obtain polynomial-time solutions for all problems in $\mathcal{NP}$!

Example 14–7: $0/1$–Integer Programming Problem
Let $\{x_1, x_2, \ldots, x_n\}$ be a set of n variables that can only take values 0 or 1. We are given a linear set of constraints. Is there a set of solutions that satisfy these constraints? For example,

$$\begin{array}{rcl} x_1 + x_2 & \geqslant & 2 \\ 4x_1 - x_3 & \geqslant & 3 \\ x_1 + 4x_2 - 3x_3 & \geqslant & 4 \end{array} \qquad \begin{array}{rcl} x_1 + x_2 & \geqslant & 2 \\ x_1 + 5x_2 - 4x_3 & \geqslant & 3 \\ x_1 - x_2 + x_3 & \geqslant & 1 \end{array}$$

$x_1 = 1,\ x_2 = 1,\ x_3 = 0.$ No Solution

(This is the decision version. There's also an optimization version where we try to make the right–hand sides maximum.)

Show that $0/1$–Integer Programming Problem is $\mathcal{NP}$–complete.

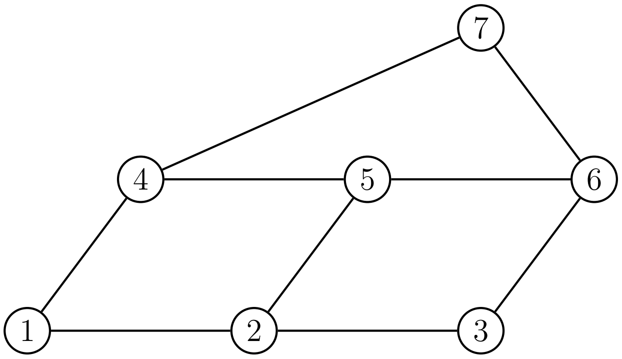
Solution: This problem is in $\mathcal{NP}$ because given a solution, we can check it in polynomial time.

If we have an algorithm that solves this problem in polynomial time, we can adapt it to solve 3–SAT problem and therefore everything else in $\mathcal{NP}$. The right-hand sides will all be 1 and we will use $1 - x_i$ for $\neg x_i$, for example,

$$\begin{array}{ccc} \underline{3\text{–SAT}} & & \underline{0/1} \\ x_1 \vee x_2 \vee x_3 & \Rightarrow & x_1 + x_2 + x_3 \geqslant 1 \\ x_2 \vee x_4 \vee \neg x_5 & \Rightarrow & x_2 + x_4 + (1 - x_5) \geqslant 1 \\ \neg x_7 \vee \neg x_8 \vee \neg x_9 & \Rightarrow & (1 - x_7) + (1 - x_8) + (1 - x_9) \geqslant 1 \end{array}$$

The reduction can be done in polynomial time therefore $0/1$–Integer Programming Problem is $\mathcal{NP}$–complete.

Example 14–8: Independent Set Problem
An Independent Set in a graph is a set of vertices that are not adjacent to any other in that set. In other words, no two of them share an edge. The problem is for given k to be able to find such a set of k vertices. For example,



$$\begin{aligned} k = 3 &\Rightarrow \{1,3,5\} \text{ or } \{2,4,6\} \\ k = 4 &\Rightarrow \{1,3,5,7\} \\ k = 5 &\Rightarrow \text{Does not exist} \end{aligned}$$

Show that Independent Set Problem is $\mathcal{NP}$–complete.

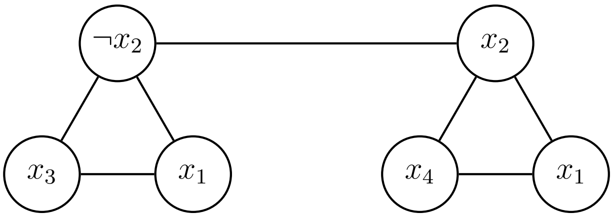
Solution: This problem is in $\mathcal{NP}$ because given a solution, we can check it in polynomial time.

Given a 3–SAT problem, we can reduce it to Independent Set problem as follows:

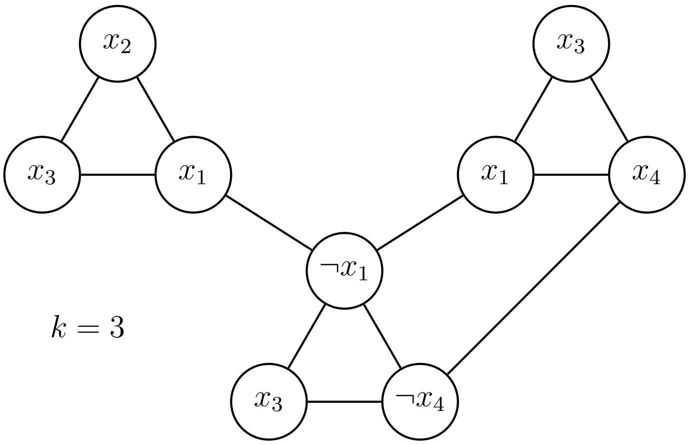
- For each parenthesis, construct three vertices and connect them to each other.
- Connect each pair of vertices that represent a variable x_i and its negation $\neg x_i$.

The number k for IS problem will be the number of parentheses in 3–SAT problem. This construction will force the algorithm to choose exactly one variable from each parenthesis as `TRUE` and not to choose logically contradictory ones from different parentheses.

For example, $\big(x_1 \vee \neg x_2 \vee x_3\big) \wedge \big(x_1 \vee x_2 \vee x_4\big)$ will be represented by:



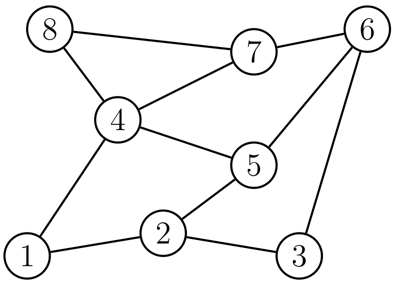
$$\big(x_1 \vee x_2 \vee x_3\big) \wedge \big(x_4 \vee x_3 \vee x_1\big) \wedge \big(\neg x_4 \vee \neg x_1 \vee x_3\big) \quad \Rightarrow$$



This transformation can be done in polynomial time therefore Independent Set Problem is $\mathcal{NP}$–complete.

Example 14–9: Vertex Cover Problem

A vertex cover in a graph is a set of vertices that cover all the edges. In other words, each edge must be connected to at least one vertex in that set. The decision problem is, given k, is there a vertex cover of k vertices? For example,



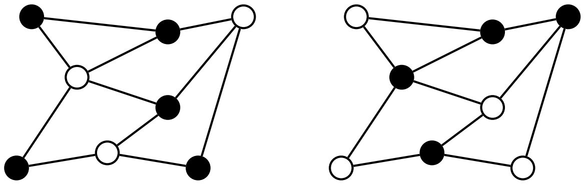
k	Vertex Cover
5	$\{1,3,5,7,8\}$
4	$\{2,4,6,7\}$
3	Does not exist

Show that Vertex Cover Problem is $\mathcal{NP}$–complete.

Solution: We can easily do this by proving:

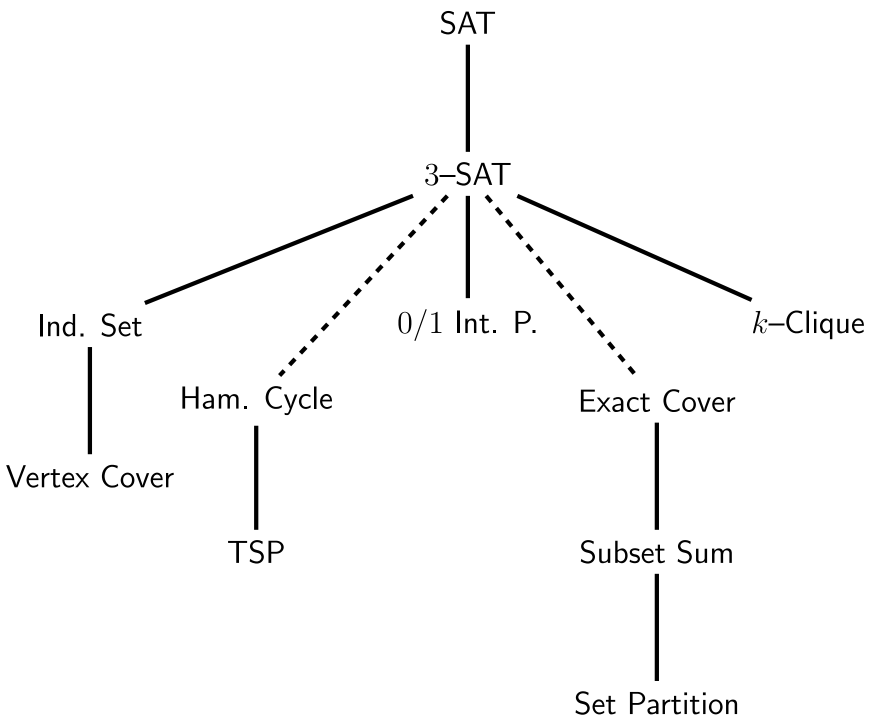
Independent Set Problem $\leqslant_P$ Vertex Cover Problem

They are almost identical problems because the vertices not in the vertex cover are an independent set and vice versa. In other words, in the set of vertices, the sets IS and VC are complements of each other. For a graph of n vertices, if we can find a vertex cover of size $n-k$, then we can find an independent set of size k.



In both graphs, black vertices denote a vertex cover and whites an independent set.

The following figure summarizes the $\mathcal{NP}$–complete problems we have studied in this chapter and the reductions we did: [3]



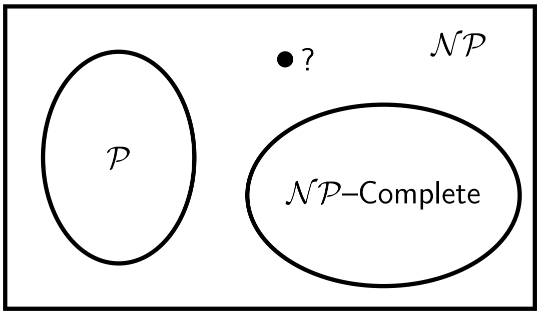
SAT $\leqslant_P$ 3–SAT $\leqslant_P$ $0/1$ Int. P. etc.

The reductions denoted by dashed lines are left as an exercise. Their proofs are more technical and longer.

There are other possible reductions between these problems. For example, k–Clique and Independent Set problems can be directly reduced to each other. In other words, the complete form of the above tree is quite complicated!

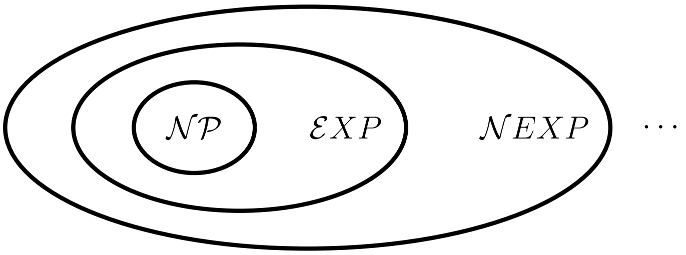
If we can find a polynomial time solution for *any* of these problems, it means we have solved all $\mathcal{NP}$–problems in polynomial time and proved that $\mathcal{P} = \mathcal{NP}$. But in spite of the efforts of many researchers since 1970s, this remains an open problem. Nobody could find such an algorithm or prove that it cannot be found.

Assuming $\mathcal{P} \neq \mathcal{NP}$ the relationship among these complexity classes are:



The dot denoted by '?' is an example of a problem that is not in $\mathcal{P}$ but also not $\mathcal{NP}$–complete. A theorem was proven in 1975 that states such problems must exist if $\mathcal{P} \neq \mathcal{NP}$, but such problems are constructed artificially. [4] We know of no natural problems in that category, called $\mathcal{NP}$–intermediate.

As the last word, don't forget that there are many complexity classes beyond $\mathcal{NP}$:



EXERCISES

Show the following:

14–1) Set–Partition Problem $\leqslant_P$ Subset–Sum Problem

14–2) Vertex Cover Problem $\leqslant_P$ k–Clique Problem

14–3) k–Clique Problem $\leqslant_P$ Vertex Cover Problem

14–4) 3–SAT Problem $\leqslant_P$ Subset–Sum Problem

14–5) Vertex Cover Problem $\leqslant_P$ Subset–Sum Problem

Show that the following problems are $\mathcal{NP}$–complete: (Note that these are all decision problems. Most have other versions where we try to optimize a quantity.)

14–6) 0–1 Knapsack Problem: There are n items with prices $p_1, p_2, \ldots, p_n$ and weights $w_1, w_2, \ldots, w_n$. Is there a subset of items such that their total value $\geqslant P$ and their total weight $\leqslant W$?

14–7) Bin Packing Problem: Given n items with sizes $s_1, s_2, \ldots, s_n$ where $0 \leqslant s_i \leqslant 1$ and N bins of unit size, will the items fit in bins?

14–8) Subgraph Isomorphism Problem: Given 2 graphs G_1 and G_2, is there a subgraph of G_2 which is isomorphic to G_1? Note that two graphs are isomorphic if they have the same number of vertices and edges and if we can find a function between them that preserves the structure.

14–9) Hamiltonian Cycle Problem on directed graphs. (Hint: Reduce 3–SAT Problem to this problem.)

14–10) 3–Coloring Problem: Given a graph G and 3 colors, is it possible to paint each vertex a color such that adjacent vertices have different colors? (Hint: Reduce 3–SAT Problem to this problem.)

ANSWERS

14–1) Given a set of integers, find their sum M. If M is odd, there is no solution. If M is even, find a subset whose sum is equal to $\frac{M}{2}$.

14–2) Given an integer k and a graph G with n vertices, consider the complement graph $\overline{G}$. Two vertices in $\overline{G}$ are connected if and only if they are not connected in G. In other words, to obtain $\overline{G}$ from G, keep all vertices, delete all edges, add all possible edges that were not in G.

Now, you can easily show that $\overline{G}$ has an $(n-k)$–clique if and only if G has a vertex cover with k vertices.

14–3) Similar to the previous solution.

14–4) Hint: Suppose there are 4 variables and 5 parentheses. Suppose x_1 is in first and fourth, $\neg x_1$ is in the fifth parenthesis. Represent x_1 by $01001\text{-}0001$ and $\neg x_1$ by $10000\text{-}0001$. (Dash is not there in the actual numbers. This is just to make representation easier.)

Add some dummy numbers. The target is $33333\text{-}1111$.

14–5) Hint: Suppose a vertex is connected to edges 1 and 3. Suppose there are 5 edges. Then represent this vertex by the number 100101 in base 4.

Target number is $k22222$ where k is also in base 4.

14–6) We can reduce Subset–Sum problem to 0–1 Knapsack Problem if we take $p_i = w_i$ and $P = W =$Target Number.

14–7) We can reduce Set–Partition problem to Bin Packing problem by dividing all numbers by the Set–Sum$/2$ and taking 2 bins.

14–8) We can reduce k–Clique Problem problem to Subgraph Isomorphism problem by taking G_1 as a complete graph with k vertices.

14–9) The necessary construction has several stages. We should represent each parenthesis in the 3–SAT problem as a vertex. There should be three arrows entering and three arrows leaving this vertex, one corresponding to each variable it contains. To force the path to leave through the correct variable's exit path, we need many additional vertices for each variable. Each variable should be either true or false. We will ensure this by choosing the right or left path through that variable's vertices. Details are left to the reader.

14–10) This reduction is also quite involved:
1) Design a construction which makes all vertices in a set the same color.
2) Design a construction which makes all vertices in a set the same color, and all vertices in a second set of a different color.
3) Make sure all vertices representing parentheses of the 3–SAT problem are colored by two colors. They will be true and false. The third one is neutral.
4) Design a construction of several vertices such that if two given vertices are the same color, a third one is also the same color. Then extend this to three given vertices.
5) Make sure that for the vertices representing variables in a parenthesis, the combination False–False–False is impossible.
You need to fill in many of the gaps.

References

[1] S.A. Cook, The Complexity of Theorem–Proving Procedures, in *Proceedings of 3rd Annual ACM Symposium on Theory of Computing,* (1971), pp. 151–158.

[2] J.E. Hopcroft, R. Motwani, and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation,* 3rd edition, Pearson, 2006.

[3] R.M. Karp, Reducibility Among Combinatorial Problems, in R. E. Miller et al. (eds) *Complexity of Computer Computations*, Plenum Press, (1972), pp. 85–103.

[4] R.E. Ladner, On the Structure of Polynomial Time Reducibility, *Journal of the Association for Computing Machinery*, Vol. 22 (1975), pp. 155–171.

[5] P. Linz, *An Introduction to Formal Languages and Automata,* 6th edition, Jones & Bartlett Learning, 2016.

[6] M. Sipser, *Introduction to the Theory of Computation,* 3rd edition, Cengage Learning, 2012.

[7] A. Turing, On Computable Numbers, with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, Vol. s2-42 (1937), pp. 230-265.

Index