# Fundamentals Of Computer Science Course Notes

Felipe Balbi

December 2, 2019

## Contents

# 1 Week 1

Learning Objectives:

- Understand logical arguments and apply basic concepts of formal proof.

## 1.1 1.101 Introduction to propositional logic

Propositional Logic is a system that deals with propositions or statements.

Below there's an example of where we can apply propositional logic to derive conclusions.

### 1.1.1 Liars And Knights

Imagine there is an island with two types of people. Liars who always tell lies and knights who always tell the truth. One an excursion, you visit the island and encounter two people, person A and person B. Person A says "at least one of us is a liar", while person B says nothing. What conclusion can you draw?

With a little logical thinking, we can conclude that person A is a knight and person B is a liar.

## 1.2 1.103 Building blocks of logic

### 1.2.1 What is a proposition?

A **proposition** is a statement that can be either **true** or **false**. It must be one or the other, never both nor neither.

Examples of proposition:

- 2 is a prime number (T)
- 5 is an even number (F)

Not a proposition:

- x is a prime number

  In this case, it can be made into a proposition by assigning a value to x.

- Are you going to school?

  Because this is a question, we can't assign a truth value to the sentence.

- Do your homework now

  Being an order, it has no truth value.

### 1.2.2 Syntaxes of the propositional logic

Propositions are denoted by capitable letters: P, Q, R, and so on.

- P = carrots are orange
- Q = I went to a party yesterday

General statements are denoted by lowercase letters: p, q, r, and so on. They carry on a logical argument, are used in proofs, called propositional variables.

### 1.2.3 Connectives: change or combine propositions

Connectives transform **atomic** propositions into **compound** propositions.

**Logical NOT ($\neg$)** $\neg p$ is true if and only if $p$ is false. Also called **negation**.

**Logical OR ($\vee$)** $p \vee q$ is true if and only if at least one of $p$ or $q$ is true or if both $p$ and $q$ are true. Also called **disjunction**.

**Logical AND ($\wedge$)** $p \wedge q$ is true if and only if both $p$ and $q$ are true and false otherwise. Also called **conjunction**.

**Logical if then ($\rightarrow$)** $p \rightarrow q$ is true if and only if either p is false or q is true. Also called **implication** or **conditional**. $p$ is called the **premise** and $q$ is the **conclusion**.

**Logical if and only if ($\leftrightarrow$)** $p \leftrightarrow q$ is true if and only if both $p$ and $q$ are true or both are false. Also called **bi-conditional**.

**Exclusive or ($\oplus$)** $p \oplus q$ is true if and only if $p$ or $q$ is true but not both.

### 1.2.4 Translation from Logical Proposition to English

Let:

$$\begin{aligned} P &= \text{I study 20 hours a week} \\ Q &= \text{I attend all the lectures} \\ R &= \text{I will pass the exam} \\ S &= \text{I will be happy} \end{aligned} \tag{1}$$

Translate the following statement to English:

- $(P \vee Q) \rightarrow (R \wedge S)$

  If **I study 20 hours a week** or **I attend all the lectures** then **I will pass the exam** and **I will be happy**.

### 1.2.5 Translation from English to Logical Proposition

Given the statement:

*If UK does not exit the EU then skilled nurses will not leave the NHS and research grants will remain intact.*

Translating to logical proposition we get:

$$\begin{aligned} P &= \text{UK exits the EU} \\ Q &= \text{Skilled nurses will leave the NHS} \\ R &= \text{Research grants will remain intact} \\ \neg P &\rightarrow \neg Q \wedge R \end{aligned} \tag{2}$$

Note that we removed any **connectives** from our propositions as that's a good practice. This makes the logical statement easier to follow.

## 1.3 1.105 Truth Table: examples

A truth table is a set of all outcomes of propositions and connectives. The number of rows in a truth table, depends on the number of given propositions. If we have $n$ propositions, our truth table will have $2^n$ rows.

### 1.3.1 Truth tables for each connective

What follows is a list of truth tables for each connective

**Negation ($\neg$)**

| $p$ | $\neg p$ |
|---|---|
| 1 | 0 |
| 0 | 1 |

**Conjunction ($\wedge$)**

| $p$ | $q$ | $p \wedge q$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

**Disjunction ($\vee$)**

| $p$ | $q$ | $p \vee q$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

**Implication ($\rightarrow$)**

| $p$ | $q$ | $p \rightarrow q$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |

**Bi-conditional ($\leftrightarrow$)**

| $p$ | $q$ | $p \leftrightarrow q$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

**Exclusive Or ($\oplus$)**

| $p$ | $q$ | $p \oplus q$ |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

### 1.3.2 Operator Precedence

When formulae are written without parenthesis, we must rely on rules of operator precedence. Logic operator precende rules are as follows:

$\neg \wedge \vee \rightarrow \leftrightarrow$

**Example** If we have the logical statement $p \rightarrow p \wedge \neg q \vee s$, we can parse it following the steps below:

$$\begin{aligned} & p \rightarrow p \wedge \neg q \vee s \\ & p \rightarrow p \wedge (\neg q) \vee s \\ & p \rightarrow (p \wedge (\neg q)) \vee s \\ & p \rightarrow ((p \wedge (\neg q)) \vee s) \\ & (p \rightarrow ((p \wedge (\neg q)) \vee s)) \end{aligned} \tag{3}$$

### 1.3.3 Constructing Truth Tables for Complex Formulae

**Example 1**

| $p$ | $q$ | $p \wedge q$ | $(p \wedge q) \rightarrow p$ |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 |

**Example 2**

| $p$ | $q$ | $q \rightarrow p$ | $p \wedge (q \rightarrow p)$ |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |

**Comparing both examples**

| $p$ | $q$ | $(p \wedge q) \rightarrow p$ | $p \wedge (q \rightarrow p)$ |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 |

## 1.4 1.202 Tautology and consistency

### 1.4.1 Tautology

A formula that is **always** true regardless of the truth value of the proposition.

| $p$ | $\neg p$ | $p \vee \neg p$ |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 0 | 1 |

### 1.4.2 Consistent

A formula that is true **at least** for one scenario. All connectives are consistent.

The formula $p \wedge \neg p$ is **inconsistent** because it can never be true. Inconsistent formulae are also called **contradictions**.

### 1.4.3 1.204 Tautology and consistency: examples

**Example 1:** $p \vee (q \wedge \neg r)$

| $p$ | $q$ | $r$ | $\neg r$ | $q \wedge \neg r$ | $p \vee (q \wedge \neg r)$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 |

This is a **Consistent** formula

**Example 2:** $(p \rightarrow q) \rightarrow (\neg q \vee r)$

| $p$ | $q$ | $r$ | $p \rightarrow q$ | $\neg q$ | $(\neg$ q $\vee$ \$) | $(p \rightarrow q) \rightarrow (\neg q \vee r)$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 |

This is a **Consistent** formula

**Example 3:** $(p \rightarrow q) \leftrightarrow (\neg p \vee q)$

| $p$ | $q$ | $p \rightarrow q$ | $\neg p$ | $(\neg p \vee q)$ | $(p \rightarrow q) \leftrightarrow (\neg p \vee q)$ |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 |

This formula is a **Tautology**

# 2 Week 2

Learning Objectives:

- Understand logical arguments and apply basic concepts of formal proof.

## 2.1 1.301 Equivalences

Formulae $A$ and $B$ are equivalent if they have identical truth tables. Equivalence is denoted by the symbol $\equiv$

In other words, $A \equiv B$ means that $A$ and $B$ have the same truth values, regardless of how variables are assigned.

One thing to note is that $\equiv$ is **NOT** a connective.

### 2.1.1 De Morgan's Laws

$$\begin{aligned} \neg(p \wedge q) &\equiv \neg p \vee \neg q \\ \neg(p \vee q) &\equiv \neg p \wedge \neg q \end{aligned} \tag{4}$$

### 2.1.2 Truth Tables

$\neg(p \wedge q) \equiv \neg p \vee \neg q$

| $p$ | $q$ | $\neg p$ | $\neg q$ | $p \wedge q$ | $\neg(p \wedge q)$ | $\neg p \vee \neg q$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 |

$\neg(p \vee q) \equiv \neg p \wedge \neg q$

| $p$ | $q$ | $\neg p$ | $\neg q$ | $p \vee q$ | $\neg(p \vee q)$ | $\neg p \wedge \neg q$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 |

$(p \to q) \equiv (\neg p \vee q) \equiv \neg(p \wedge \neg q)$

| $p$ | $q$ | $\neg p$ | $\neg q$ | $p \to q$ | $\neg p \vee q$ | $\neg(p \wedge \neg q)$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 |

**Contrapositive:** $(p \to q) \equiv (\neg q \to \neg p)$

| $p$ | $q$ | $\neg p$ | $\neg q$ | $p \to q$ | $\neg q \to \neg p$ |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 |

## 2.2 1.304 First-order logic

### 2.2.1 Important Notions

- **Predicates** describe properties of objects

  A simple example could be `odd(3)`. Here we're applying the predicate `odd` to the object `3`. When arguments are applied to predicates, they become propositions and connectives for propositional logic can be employed in the usual manner:

$$Odd(3) \wedge Prime(3) = T \tag{5}$$

- **Quantifiers** allow reasoning on multiple objects

  The objects from a quantified statement are chosen from a *Domain*.

  - **Existential Quantifier** $\exists$

    We use it as follows: $\exists x$ some formula.

    When proving a formula based on the existential quantifier, it is enough to find **one** element which makes the formula true. In other words, existentially quantified statements are **false** unless there is a positive example.

  - **Universal Quantifier** $\forall$

    We use it as follows: $\forall x$ some formula.

    In order to satisfy the formula, we must prove that **every** x satisfies the formula.

    Note that a single counterexample is enough to disprove a universally quantified statement. In other words, universally quantified statements are **true** unless there is a false example.

### 2.2.2 Translations English - Logic

"All P's are Q's" translates into $\forall x(P(x) \to Q(x))$

"No P's are Q's" translates into $\forall x(P(x) \to \neg Q(x))$

"Some P's are Q's" translates into $\exists x(P(x) \wedge Q(x))$

"Some P's are not Q's" translates into $\exists x(P(x) \wedge \neg Q(x))$

### 2.2.3 Quantifiers to connectives

- Existential Quantifier

$\exists x P(x)$ and domain $D = \{x_1, x_2, \ldots, x_n\}$. This is equivalent to saying $P(x_1) \vee P(x_2) \vee \ldots \vee P(x_n)$

- Universal Quantifier

$\forall x P(x)$ and domain $D = \{x_1, x_2, \ldots, x_n\}$. This is equivalent to saying $P(x_1) \wedge P(x_2) \wedge \ldots \wedge P(x_n)$

### 2.2.4 Negation of Quantifiers

- Existential Quantifier

$$\begin{aligned} \neg\exists x P(x) &\equiv \neg(P(x_1) \vee P(x_2) \vee \ldots \vee P(x_n)) \\ &\equiv \neg P(x_1) \wedge \neg P(x_2) \wedge \ldots \wedge \neg P(x_n)) \\ &\equiv \forall x \neg P(x) \end{aligned} \tag{6}$$

- Universal Quantifier

$$\begin{aligned} \neg\forall x P(x) &\equiv \neg(P(x_1) \wedge P(x_2) \wedge \ldots \wedge P(x_n)) \\ &\equiv \neg P(x_1) \vee \neg P(x_2) \vee \ldots \vee \neg P(x_n)) \\ &\equiv \exists x \neg P(x) \end{aligned} \tag{7}$$

**Example**

$$\begin{aligned} \neg(\forall x(P(x) \to Q(x)) &\equiv \exists x \neg(P(x) \to Q(x)) \\ &\equiv \exists x \neg(\neg P(x) \vee Q(x)) \\ &\equiv \exists x(\neg\neg P(x) \wedge \neg Q(x)) \\ &\equiv \exists x(P(x) \wedge \neg Q(x)) \end{aligned} \tag{8}$$

# 3 Week 3

Learning Objectives:

- Correctly follow a sequence of justified steps to reach a conclusion statement.
- Prove a conclusion statement by first assuming it is false.
- Describe inductive steps.
- Understand logical arguments and apply basic concepts of formal proof.

## 3.1 2.01 What is a proof?

A proof is a sequence of logical statements that explains why a statement is true. Rosen's book defines a proof as follows:

> A proof is a valid argument that establishes the truth of a mathematical statement.

We need proofs to establish general truths about conjectures. For example a computer cannot confirm that **all** numbers have a certain property. Well, considering that numbers are infinite and computers work with finite amounts of memory, a computer will not be able to answer that question.

Given a theorem, often there are many ways in which we can prove it. There are commonly used proof techniques which we learn about.

## 3.2 2.101 Direct proof

A direct proof exploits definitions and other mathematical theorems. It arrives at the desired statement by employing valid logical steps.

A direct proof is:

- Easy because there is no particular technique is used
- Not easy because the starting point is not obvious
- Know your definitions
- Allowed to use any theorem, axiom, logic, etc

### 3.2.1 Example 1:

**Theorem 1.** *If $n$ and $m$ are even numbers, then $n+m$ is also even*

*Proof.* What does even mean?

If an integer is even, it is twice another integer.

$$n = 2k$$
$$m = 2l$$

$k$ and $l$ are integers

$$\begin{aligned} n + m &= 2k + 2l \\ &= 2(k + l) \end{aligned}$$

Let $k + l = t$,

$$n + m = 2t$$

$\therefore n + m$ is even. □

### 3.2.2 Example 2:

**Theorem 2.** *$\forall n \in \mathbb{N}, n^2 + n$ is even.*

*Proof.* If $n$ is even, then $n = 2k$.

$$n^2 + n = (2k)^2 + 2k \,\text{Even}$$

If $n$ is odd, then $n = 2k + 1$.

$$\begin{aligned} n^2 + n &= (2k+1)^2 + 2k + 1 \\ &= (2k)^2 + 2 \cdot 2k + 1^2 + 2k + 1^2 \\ &= 4k^2 + 6k + 2 \,\text{Even} \end{aligned}$$

□

## 3.3 2.102 Direct proof examples

### 3.3.1 Example 1:

**Theorem 3.** *If $a < b < 0$, then $a^2 > b^2$*

*Proof.* Assume $a < b$ and $a < 0$. Multiplying both sides of the inequality by $a$ gives:

$$\begin{aligned} a \cdot a &< b \cdot a \\ a^2 &> b \cdot a \end{aligned}$$

Assume $a < b$ and $b < 0$. Multiplying both sides of the inequality by $b$ gives:

$$\begin{aligned} a \cdot b &< b \cdot b \\ a \cdot b &> b^2 \end{aligned}$$

By the commutative property of multiplication we know that $a \cdot b = b \cdot a$, therefore:

$$\begin{aligned} a^2 > a \cdot b > b^2 \\ \therefore a^2 > b^2 \end{aligned}$$

□

### 3.3.2 Example 2:

**Theorem 4.** *$\forall x \in \mathbb{N}, 2x^3 + x$ is a multiple of 3.*

*Proof.* Factorizing $2x^3 + x$ gives $x(2x^2 + 1)$.

If $x$ is a multiple of 3, the proof is complete.

If $x = 3k + 1$, then:

$$\begin{aligned} x(2x^2 + 1) &= (3k+1)[2(3k+1)^2 + 1] \\ &= (3k+1)[2(9k^2 + 6k + 1) + 1] \\ &= (3k+1)(18k^2 + 12k + 3] \\ &= 3(3k+1)((6k^2 + 4k + 1) \end{aligned}$$

If $x = 3k + 2$, then:

$$\begin{aligned} x(2x^2+1) &= (3k+2)[2(3k+2)^2+1] \\ &= (3k+2)[2(9k^2+12k+4)+1] \\ &= (3k+2)(18k^2+24k+9) \\ &= 3(3k+2)(6k^2+8k+3) \end{aligned}$$

□

## 3.4 2.202 Proof by contradiction

Proof by contradiction is also referred to as *indirect proof*. It follows a simple structure to prove that statement **A** is *true*.

We start by assuming **A** to be *false*, we follow just like a direct proof by employing mathematical definitions, theorems, axioms and logical steps until we arrive at a statement which **contradicts** our original assumption. This would show our original assumption to be incorrect. Therefore, if our assumption is **not** false, then it can only be true.

### 3.4.1 Example 1:

**Theorem 5.** *The square-root of two,* $\sqrt{(}2)$ *is irrational*

*Proof.* Assume $\sqrt{(}2)$ is rational. This means it can be written as a fraction, in lowest terms, of the form $\frac{p}{q}$ for $p, q \in \mathbb{N}$, $q \neq 0$.

If $\frac{p}{q}$ is in lowest terms, it means the fraction cannot be further simplified. Therefore, we have:

$$\begin{aligned} \sqrt{(}2) &= \frac{p}{q} \\ \left(\sqrt{(}2)\right)^2 &= \left(\frac{p}{q}\right)^2 \\ 2 &= \frac{p^2}{q^2} \\ 2q^2 &= p^2 \end{aligned}$$

From this, we can see that $p$ must be even. Which means $p = 2k$. Therefore:

$$\begin{aligned} 2q^2 &= p^2 \\ 2q^2 &= (2k)^2 \\ 2q^2 &= 4k^2 \\ q^2 &= 2k^2 \end{aligned}$$

From this, we can see that $q$ must also be even. Which means our fraction $\frac{p}{q}$ cannot be in lowest terms. Therefore, $\sqrt{(}2)$ cannot be a rational number, so it is irrational. □

### 3.4.2 Example 2:

**Theorem 6.** *There is an infinite number of prime numbers.*

*Proof.* Assume there are finitely many prime numbers. Let the set of prime numbers be $P = \{p_1, p_2, \ldots, p_n\}$. Let $N = (p_1 \cdot p_2 \cdot \ldots \cdot p_n) + 1$.

If we divide $N$ by any of the prime numbers in our list of prime numbers, it will have a remainder of 1. Therefore, $N$ is, itself, a prime number. □

## 3.5 2.203 Proof by contraposive

This technique exploits equivalent classes of logical statements. Let us remember that $a \to b \equiv \neg b \to \neg a$.

In some cases, when we need to prove $a \to b$, it may be easier to prove its contrapositive $(\neg b \to \neg a)$ is true.

### 3.5.1 Example 1:

**Theorem 7.** $\forall n \in \mathbb{N}, Odd(n^3 + 1) \to Even(n)$

*Proof.* By means of the contrapositive $\forall n \in \mathbb{N}, Odd(n) \to Even(n^3 + 1)$.

$$n = 2k + 1 \forall k \in \mathbb{N}$$

$$\begin{aligned} n^3 + 1 &= (2k+1)^3 + 1 \\ &= (2k+1)(2k+1)(2k+1) + 1 \\ &= 8k^3 + 12k^2 + 6k + 2 \\ &= 2(4k^3 + 6k^2 + 3k + 1) \end{aligned}$$

□

### 3.5.2 Example 2:

**Theorem 8.** *Suppose* $x, y \in \mathbb{R}$, $y^3 + yx^2 \leq x^3 + xy^2 \to y \leq x$

*Proof.* By contrapositive $y > x \to y^3 + yx^2 > x^3 + xy^2$.

Assuming $y > x$, we know that $y - x > 0$. Let us multiply both sides of the inequality by $x^2 + y^2$. Therefore:

$$\begin{aligned} (y^2 + x^2)(y - x) &> 0(y^2 + x^2) \\ y^3 + yx^2 - xy^2 - x^3 &> 0 \\ y^3 + yx^2 &> x^3 + xy^2 \end{aligned}$$

□

# 4 Week 4

Learning Objectives:

- Correctly follow a sequence of justified steps to reach a conclusion statement.
- Prove a conclusion statement by first assuming it is false.
- Describe inductive steps.
- Understand logical arguments and apply basic concepts of formal proof.

## 4.1 2.301 Proof by induction

Mathematical induction is a useful proof method that has several steps that must be followed. We can consider mathematical induction as a row of standing dominoes and we want to prove that all dominoes fall.

In order to prove that all dominoes fall, we must first and foremost prove that the first domino falls. After that we prove that if one domino falls, the next one must also fall.

Mathematically, if $P(0)$ is true and $\forall k \in \mathbb{N} P(k) \rightarrow P(k+1)$, then we can conclude that $\forall n \in \mathbb{N} P(n)$ is true.

Proof by induction has three important steps:

- The *Base Case* or *Basis*

  Here we prove that $P(0)$ is true. This allows us to prove that the theorem **starts** true.

- The *Inductive Step*

  Here we prove that $P(k) \rightarrow P(k+1)$. Note that we never assume this to be true. We must always carefully prove it. Because we're trying to prove an implication, we assume $P(k)$ to be true and prove $P(k+1)$ to be true when $P(k)$ is true. The reason for this is that if $P(k)$ is false, then the implication is true anyway. The assumption that $P(k)$ is true is called the *Inductive Hypothesis*.

- The *Conclusion by Induction*

  We finish the proof by writing $\therefore \forall n \in \mathbb{N} P(n)$ is true.

It's common practice to end a proof with the $\square$ symbol. Referred to as **QED** (from Latin *quod erat demonstrandum*).

## 4.2 2.303 Example of a correct proof

### 4.2.1 $\sum_{i=0}^{n-1} 2^i = 2^n - 1$

**Theorem 9.** *The sum of the first n powers of 2, is* $2^n - 1$.

*Proof.* Let $P(n) = 2^0 + 2^1 + \ldots + 2^{n-1} = 2^n - 1$, prove that $P(n)$ is valid for all $n$.

Basis: Prove that $P(1)$ is true.

$$\begin{aligned} P(1) &= 2^{1-1} \\ &= 2^0 \\ &= 1 \\ &= 2^1 - 1 \end{aligned}$$

Indutive Step: Prove that $P(k) \to P(k+1)$ is true.

Assuming $P(k) = 2^0 + 2^1 + \ldots + 2^{k-1} = 2^k - 1$ to be true, prove that $P(k+1) = 2^0 + 2^1 + \ldots + 2^{k-1} + 2^k = 2^{k+1} - 1$ is also true.

$$P(k+1) = 2^0 + 2^1 + \ldots + 2^{k-1} + 2^k$$

By Inductive Hypothesis we know that $2^0 + 2^1 + \ldots + 2^{k-1} = 2^k - 1$, therefore:

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

Conclusion: $P(k+1)$ is true, $\therefore \forall n \in \mathbb{N} 2^n - 1$ is true. □

#### 4.2.2 $\forall n n < 3^n$

**Theorem 10.** *$n < 3^n$, for all $n \in \mathbb{N}$*

*Proof.* Let $P(n) = n < 3^n$, prove by induction that $P(n)$ is true for all $n$.

Basis: Prove $P(0)$ is true.

$$\begin{aligned} P(0) &= 0 < 3^0 \\ &= 0 < 1 \end{aligned}$$

Inductive Step: Prove $P(k) \to P(k+1)$.

Assuming $P(k) = k < 3^k$ to be true, prove that $P(k+1) = (k+1) < 3^{k+1}$ is true.

$$\begin{aligned} k &< 3^k \\ k+1 &< 3^k + 1 \\ k+1 &< 3^k + 1 < 3^k + 3^k + 3^k \\ k+1 &< 3^k + 1 < 3 \cdot 3^k \\ k+1 &< 3^k + 1 < 3^{k+1} \\ k+1 &< 3^{k+1} \end{aligned}$$

Conclusion: $P(k+1)$ is true, $\therefore \forall n \in \mathbb{N} n < 3^n$ is true. □

## 4.3 2.305 Example of an incorrect proof

We're going to see how easy it is to make a mistake in a proof if we don't follow the steps correctly.

### 4.3.1 $n + 1 < n \forall n \in \mathbb{N}$

**Theorem 11.** *$n + 1 < n$, for all $n \in \mathbb{N}$.*

*Proof.* INCORRECT!!!

Let $P(n) = n + 1 < n \forall n \in \mathbb{N}$.

Prove $P(k) \rightarrow P(k + 1)$. Assuming $P(k)$ is true, so $k + 1 < k$. Show $P(k + 1)$ is true. Adding 1 to both sides of the inequality we get $k + 1 + 1 < k + 1$. Let $l = k + 1$ we get $l + 1 < l$, so $P(k + 1)$ is also true. Therefore $P(n)$ is true. □

In this proof, we didn't prove the base case, so the proof in invalid.

## 4.4 2.401 Conclusion

We have learned about several powerful proof techniques. We explored Proof by Induction, which exploits the fact that natural numbers are like a chain.

We have also seen how contrapositive proofs are, sometimes, easier then proving the original statement. We have also witnessed how proofs can go wrong if we miss an important step.

# 5 Week 5

Learning Objectives

- Explore finite or countable discrete structures in the context of computer science.
- Consider how different rules can be applied to appreciate the number of possible outcomes for an event.
- Explore relationships between sets and elements within or across sets.
- Consider how elements in a set can be counted.

## 5.1 3.01 Introduction

During this topic, we study key principles in counting. We study the Pigeon-hole principle and learn to apply to prove theorems.

## 5.2 3.101 Counting

How many outfits can we pick from a collection of 5 pairs of trousers and 7 shirts? Essentially this translates to:

$$
\begin{aligned}
&\binom{7}{1} \cdot \binom{5}{1} \\
&\qquad = \frac{7!}{1! \cdot (7-1)!} \cdot \frac{5!}{1! \cdot (5-1)!} \\
&\qquad = \frac{7 \cdot 6!}{6!} \cdot \frac{5 \cdot 4!}{4!} \\
&\qquad = 7 \cdot 5 \\
&\qquad = 35
\end{aligned}
$$

### 5.2.1 Product Rule

The product rule says that if a job can be split into two separate tasks, if there are $n$ ways of doing task 1 and $m$ ways of doing task 2 then the job can be done in $n \cdot m$ ways.

A generalization of this is to state that if there are $k$ tasks and each task can be achieved in $n_i$ ways, then there are $n_1 \cdot n_2 \cdot \ldots \cdot n_k$ ways of achieving the task.

**Example 1** How many strings of length 5 can we make with uppercase English letters?

We have 26 uppercase letters in the English alphabet and there are no restrictions to repetition. For each letter we have 26 options, therefore we can make as many as $26 \cdot 26 \cdot 26 \cdot 26 \cdot 26 = 26^5 = 11881376$.

**Example 2** How many strings of length 5 can we make with 3 uppercase English letters and 2 digits?

This is going to be $26 \cdot 26 \cdot 26 \cdot 10 \cdot 10 = 26^3 \cdot 10^2 = 1757600$

### 5.2.2 Sum Rule

The sum rule states that if a job can be done in $n$ ways **or** $m$ ways, then it can be done in $n + m$ ways.

**Example 1** A teacher is choosing a student to be her assistent from 5 different classes. The classes contain 28, 21, 24, 25, and 27 students. How many possible ways are there to pick an assistant?

There are $28 + 21 + 24 + 25 + 27 = 125$ ways of picking an assistant.

## 5.3 3.102 Complex counting

Continuing with counting, looking at more advanced techniques.

### 5.3.1 Example 1

For most accounts, you need to choose a password. In this example, the password must be five to seven characters long. Each drawn from uppercase letters or digits. The password must contain at least one upper case letter.

Let's split this work by length:

| Passwords | Length 5 | Length 6 | Length 7 |
|---|---|---|---|
| All passwords (1) | $36^5$ | $36^6$ | $36^7$ |
| No Letters (2) | $10^5$ | $10^6$ | $10^7$ |
| Valid Passwords (1 - 2) | 60 366 176 | 2 175 782 336 | 78 354 164 096 |
| | | **Total** | 80 590 312 608 |

### 5.3.2 Subtraction Rule

The subtraction rule applies when lists have items in common. This rule is also known as *Inclusion-Exclusion Principle*.

This rule states that if a choice can be made from two lists containing $n$ and $m$ items, then the number of ways to make a choice from these two lists is $n + m -$ items in common.

**Example 1** How many integers less than 100 are divisible by either 2 or 3. In other words, we're talking about the cardinality of the union of the set of numbers divisible by 2 and less than 100 and the set of numbers divisible by 3 and less than 100.

However, this would be cumbersome to calculate. A simpler way is to first calculate how many numbers between 1 and 99 are divisible by 2 using $\lfloor \frac{99}{2} \rfloor = 49$. Similarly, we can calculate how many numbers between 1 and 99 are divisible by 3 using $\lfloor \frac{99}{3} \rfloor = 33$.

We must remember to decrement numbers that are divisible by both 2 and 3. Such numbers are divisible by 6, therefore $\lfloor \frac{99}{6} \rfloor = 16$.

So the answer to our original question is $49 + 33 - 16 = 66$.

## 5.4 3.201 The Pigeonhole Principle

The Pidgeonhole Principle states that if $n$ items are put into $m$ containers, with $n > m$, then at least one container must contain more than one item. The Pidgeonhole Principle is also know as *Dirichlet's drawer principle*.

The generalized pigeonhole principle states that:

**Theorem 12.** *If there are $N$ objects to be placed in $k$ boxes, then at least one box contains the $\lceil \frac{N}{k} \rceil$ objects.*

*Proof.* **By contradiction**

Assume **none** of the boxes contains more than $\lceil \frac{N}{k} \rceil - 1$ objects. Since we have $k$ boxes, we can conclude that Number of Objects $\leq k(\lceil \frac{N}{k} \rceil - 1) < k(\frac{N}{k} + 1 - 1) = N$.

From that we conclude that Number of Objects $\leq N$, which contradicts our original statement that there are exactly $N$ objects. □

### 5.4.1 Example 1

How many cards from a standard deck of 52 cards must be selected to guarantee that 5 cards are from the same suit?

There are 4 suits in a standard deck of cards. If we pick 16 cards and spread them evenly among the suits, we will end up with 4 cards for each suit. At the moment we pick the $17^{\text{th}}$ card, it must go to one of the 4 suits, therefore giving 5 cards from the same suit.

We can verify this with $\lceil \frac{17}{4} \rceil = 5$.

## 5.5 3.202 The Pigeonhole Principle: examples

### 5.5.1 Example 1

In a group of 4 integers, show that there are at least two with the same remainder when divided by 3.

There are exactly three possible remainders when dividing numbers by three. Either the number is divisible by 3, giving a remainder of zero, or it is 1 above a multiple of three, or 2 above a multiple of three.

In any group of 4 integers, we will have at least two numbers with the same remainder when divided by three. This means that we have three boxes (the three possible remainders) and 4 objects (our 4 randomly selected integers). By the pidgeonhole principle, we know that at least one box will contain more than one object.

### 5.5.2 Example 2

A bag contains 7 blue balls and 4 red balls, how many must be selected to guarantee that three balls are of the same color.

In this case, there are 2 boxes (the colors) and 11 objects. In the worst case, we pick colors evenly. Assuming we have picked 4 balls (2 blue and 2 red), when we pick the $5^{\text{th}}$ ball, it must be either blue or red, therefore giving us our 3 balls of the same color.

In other words, we want to find the number which satisfies $\lceil \frac{x}{2} \rceil = 3$.

### 5.5.3 Example 3

Select 5 integers from the set $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$; show that at least two integers add up to 9.

The pairs making up 9 are $(1, 8)$, $(2, 7)$, $(3, 6)$, $(4, 5)$. If we label those pairs $A$, $B$, $C$, $D$, we can see that all numbers in the set belong to one of 4 boxes.

There are 4 boxes and 5 objects, therefore at least one box will have more than 1 object.

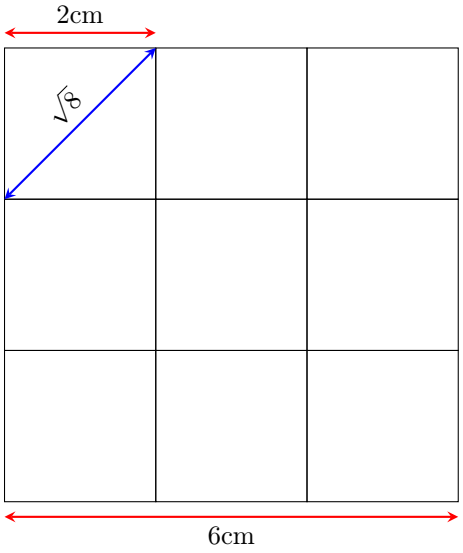
### 5.5.4 Example 4

There are $n$ people in the room; every pair is either friends or not friends. Show that there are at least two people with the same number of friends.

We know that there are $n$ in the room and the amount of friends people can have are limited to $1, 2, 3, \ldots, n-1$ or $0, 1, 2, \ldots, n-2$. In both of these cases we will have $n-1$ boxes and $n$ people. Consequently, at least one box will have more than one person.

### 5.5.5 Example 5

Show that if there are 10 dots on a square of $6cm \times 6cm$, there are at least two dots within $\sqrt{8}$ cm.




We can see that a $6cm \times 6cm$ square divided into 9 equal smaller squares of $2cm \times 2cm$. The smaller squares have a hypotenuse of $hyp = \sqrt{2^2 + 2^2} = \sqrt{8}$ (indicated by the blue line).

We have 9 boxes that are $\sqrt{8}$ apart, but have 10 objects. Therefore at least two will be within $\sqrt{8}$ distance from each other.

# 6 Week 6

Learning Objectives

- Explore finite or countable discrete structures in the context of computer science.
- Consider how different rules can be applied to appreciate the number of possible outcomes for an event.
- Explore relationships between sets and elements within or across sets.
- Consider how elements in a set can be counted.

## 6.1 3.301 Permutations

Permutation relates to the arrangement of objects where order does **not** matter. For example, how many ways are there for 5 people to form a queue? For the first position in the queue, we can have any of the five people, for second position we can have one out of four people; for the third position, we can have one out of three people and so on. The answer here is that there are $5! = 120$ ways for 5 people to form a queue.

**Definition 1** (Permutation)**.** *A permutation of a set of distinct objects is an ordered arrangement of these objects.*

There are $n!$ permutations of $n$ objects. This is a simplification of the definition of Permutation of $n$ objects taken $r$ at a time. If we set $k = r$ we get:

$$\begin{aligned} {}^{n}P_{n} &= \frac{n!}{(n-n)!} \\ &= \frac{n!}{0!} \\ &= n! \end{aligned}$$

The $r$-permutation of a set of $n$ elements is denoted by ${}^{n}P_{r} = P(n, r)$.

**Theorem 13.** *For two integers $n, r, 0 \leq r \leq n$. There are:*

$${}^{n}P_{r} = P(n, r) = n(n-1) + \ldots + (n-r+1) = \frac{n!}{(n-r)!}$$

## 6.2 3.304 Combinations

Combination relates to the arrangement of objects where **does** matter. For example, if we have 4 animals (mouse, cat, dog, rabbit) and we want to take a side-by-side photo of two animals, how many ways can we choose two animals? Note that photos with the same animals in different order count as equivalent $((cat, dog) \equiv (dog, cat))$.

**Definition 2** (Combination)**.** *A combination of a set of distinct objects is an unordered arrangement of these objects.*

There is only **one** combination of $n$ elements. This is because upon shuffling, only the order of elements change, however with combinations the **order does not matter**.

Similarly to Permutations, an $r$-combination of elements of a set is an **unordered** selection of $r$ elements from this set and is denoted by ${}^{n}C_{r} = C(n, r)$.

**Theorem 14.** *For two integers $n, r, 0 \leq r \leq n$. There are:*

$${}^{n}C_{r} = C(n, r) = \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

### 6.2.1 Example 1

How many hands of 7 cards can be dealt from a standard deck of 52 cards? First we need to figure out if the order matters. In this case it doesn't because being dealt (5, 7) is equivalent to being dealt (7, 5).

$$\begin{aligned}
{}^{n}C_{r} &= \binom{n}{r} \\
&= \binom{52}{7} \\
&= \frac{52!}{7!(52-7)!} \\
&= \frac{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48 \cdot 47 \cdot 46 \cdot 45!}{7!45!} \\
&= \frac{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48 \cdot 47 \cdot 46 \cdot 45!}{7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1} \\
&= 133\,784\,560
\end{aligned}$$

What if we're dealing 45 card hands, instead?

$$\begin{aligned}
{}^{n}C_{r} &= \binom{n}{r} \\
&= \binom{52}{45} \\
&= \frac{52!}{45!(52-45)!} \\
&= \frac{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48 \cdot 47 \cdot 46 \cdot 45!}{45!7!} \\
&= \frac{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48 \cdot 47 \cdot 46 \cdot 45!}{7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1} \\
&= 133\,784\,560
\end{aligned}$$

This shows that $C(n, r) = C(n, n - r)$.

### 6.2.2 Example 2

How many ways are there to choose 11 players from a group of 16 in order to form a team? First we need to decide if the order matters. In case, it doesn't. Therefore it's a combination.

$$
\begin{aligned}
{}^{n}C_r &= \binom{n}{r} \\
&= \binom{16}{11} \\
&= \frac{16!}{11!(16-11)!} \\
&= \frac{16 \cdot 15 \cdot 14 \cdot 13 \cdot 13 \cdot 12 \cdot 11!}{11! \cdot 5!} \\
&= \frac{16 \cdot 15 \cdot 14 \cdot 13 \cdot 13 \cdot 12}{5!} \\
&= 4368
\end{aligned}
$$

### 6.2.3 Example 3

How many binary words of length 8 contain equal number of zeroes and ones?

Does the order matter? No. Therefore it's a combination.

$$
\begin{aligned}
{}^{n}C_r &= \binom{n}{r} \\
&= \binom{8}{4} \\
&= \frac{8!}{4!(8-4)!} \\
&= \frac{8!}{4!4!} \\
&= \frac{8 \cdot 7 \cdot 6 \cdot 5 \cdot 4!}{4!4!} \\
&= \frac{8 \cdot 7 \cdot 6 \cdot 5}{4!} \\
&= 70
\end{aligned}
$$

### 6.2.4 Example 4

How many binary words of length 8 contain at most 3 ones?

At most means 0, 1, 2, or 3. This can be solved as a sum of combinations:

$$
\begin{aligned}
&= \binom{8}{0} + \binom{8}{1} + \binom{8}{2} + \binom{8}{3} \\
&= \frac{8!}{0!8!} + \frac{8!}{1!7!} + \frac{8!}{2!6!} + \frac{8!}{3!5!} \\
&= 1 + 8 + 28 + 56 \\
&= 93
\end{aligned}
$$

Note that $0! = 1$.

### 6.2.5 Example 5

How many binary words of length 8 contain at least 5 ones?

At least means 5, 6, 7 or 8. This can be solved as a sum of combinations:

$$\begin{aligned}
&= \binom{8}{5} + \binom{8}{6} + \binom{8}{7} + \binom{8}{8} \\
&= \frac{8!}{5!3!} + \frac{8!}{6!2!} + \frac{8!}{7!1!} + \frac{8!}{8!0!} \\
&= 56 + 28 + 8 + 1 \\
&= 93
\end{aligned}$$

# 7 Week 7

Learning Objectives

- Understand the basic terminologies of automata theory.
- Describe finite automata and what it can represent.
- Build deterministic and nondeterministic finite automata.
- Understand and apply various concepts in automata theory, such as deterministic automata, regular languages and context-free grammar.

## 7.1 4.01 Introduction

We will study simple mathematical machines knows as *Automata*. The next few topics will introduce the prerequisites for understanding automata and later we will learn about how these machines work and process input data.

Riddle:

> There is a farmer who has a mouse, a cat, and a loaf of bread. He has to cross the river from the north to the south by a small boat. He can take up to one of his possessions with him on the boat. The boat cannot operate without the farmer. The cat and the mouse cannot be left alone, as the cast will eat the mouse, and the mouse cannot be left alone with the loaf of bread as it will eat it. How can the farmer cross deliver without losing one of his possessions?

## 7.2 4.101 Basic definitions, letters, strings

An alphabet is denoted by the capital greek letter sigma $\Sigma$. If $\Sigma = \{0, 1\}$, we have a **binary alphabet**. If $\Sigma = \{a, b, \ldots, z\}$, then we have a collection of lower case letters.

A *string* or *word* is a finite sequence of letters drawn from an alphabet $\Sigma$. Empty strings denoted by $\varepsilon$ are strings with zero occurrences of letters from any alphabet.

The length of string $x$ is denoted by $|x|$. For example if $x = 01110101$ then $|x| = 8$. Similarly $|\text{Life is good}| = 12$

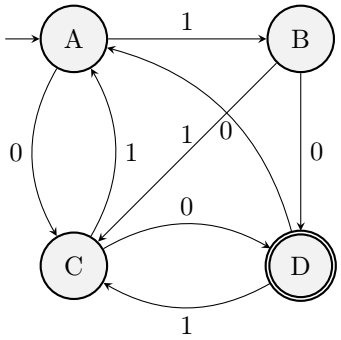



Figure 1: Finite Automaton Example

Given an alphabet $\Sigma$ there are a few different things we can do with it:

- The set of **all strings** composed of letters in $\Sigma$ is denoted by $\Sigma^*$

  Note that this is an infinite sequence of possible strings. For example if $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\varepsilon\ 0, 1, 00, 01, 10, 11, \ldots\}$

- The set of **all non-empty strings** composed of letters in $\Sigma$ is denoted by $\Sigma^+$

  Note that this is also an infinite sequence of possible strings. For example if $\Sigma = \{0, 1\}$, then $\Sigma^* = \{0, 1, 00, 01, 10, 11, \ldots\}$

- The set of **all strings of length *k*** composed of letters in $\Sigma$ is denoted by $\Sigma^k$

  Note that this is a **finite** sequence of possible strings. For example if $\Sigma = \{0, 1\}$, then $\Sigma^2 = \{00, 01, 10, 11\}$. For any alphabet $\Sigma$, $|\Sigma^k| = |\Sigma|^k$[1]

A language is a collection of strings over an alphabet. For example the language of palindromes over the binary alphabet $\Sigma = \{0, 1\}$ is $\{\varepsilon, 0, 1, 00, 11, 000, 010, 101, 111, \ldots\}$.

### 7.2.1 Examples

**If $\Sigma = \{a, b, c\}$ then what is $\Sigma^2$?** $\Sigma^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$

**If $\Sigma = \{a, b, c\}$ then what is $\Sigma^1$?** $\Sigma^1 = \{a, b, c\}$

Note that $\Sigma^1 \neq \Sigma$. Elements in $\Sigma$ are called **symbols** while elements in $\Sigma^1$ are called **strings**, the strings just happen to have length 1.

## 7.3 4.103 What is an automaton?

A *Finite Automaton* is a simple mathematical machine. It's a mathematical model of how computations are performed withi **limited memory** space. This machine contains input and output (Reject or Accept).

[1]The length of the set of all strings of length $k$ from alphabet $\Sigma$ is equal to the length of the alphabet $\Sigma$ to the power $k$.

Each circle in this image is a **state** in the automaton. State $A$ is what we call the **initial state**. An initial state will always have an arrow coming from nowhere. Each of the arrows denote state **transitions** and their labels come from the alphabet dictating what to do next.

For example, if we are at state $A$ and we read a 1, then we go to state $B$. Similarly, if we are at state $A$ and read a 0, then we go to state $C$.

In this diagram, $D$ is referred to as an **accepting state**. accept states will always be draw as a double circle. What they mean is that if the computation ends at an accept state, then the machine outputs $ACCEPT$.

### 7.3.1 Formal definition of Finite Automaton

An automaton $M$ is a 5-tuple[2] $(Q, \Sigma, \delta, q_0, F)$ where:

- $Q$ is a finite set called the **states**
- $\Sigma$ 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**

Using our previous example in figure 1, let's formalise its definition:

- $Q = \{A, B, C, D\}$
- $\Sigma = \{0, 1\}$
- $q_0 = \{A\}$
- $F = \{D\}$
- $\delta$ can be represented by the following state transition table

| $\delta$ | 0 | 1 |
|---|---|---|
| A | C | B |
| B | D | C |
| C | D | A |
| *D | A | C |

## 7.4 4.201 Finite automata: example

We look at an example of an automaton with 5 states and 2 inputs. We will see if a given input is accepted of rejected by the automaton. Figure 2 has a diagram of this automaton.

- $Q = \{A, B, C, D, E\}$
- $\Sigma = \{0, 1\}$
- $q_0 = A$
- $F = \{D, E\}$

[2] A sequence of 5 elements.

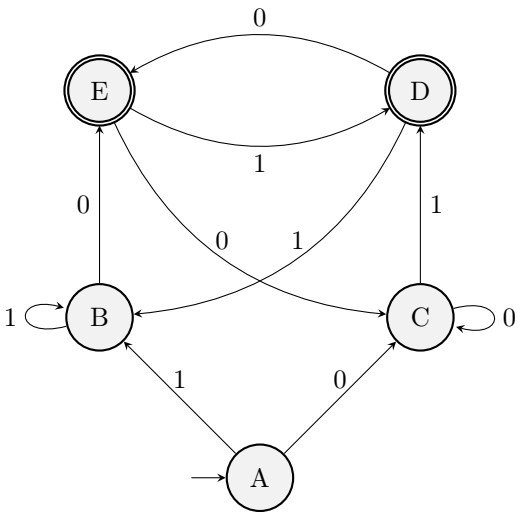



Figure 2: Automaton with 2 inputs

- $\delta$ is represented by the table below

| | 0 | 1 |
|---|---|---|
| A | C | B |
| B | E | B |
| C | C | D |
| D | E | B |
| E | C | D |

We give the input 10011 to this automaton and start computation from state $A$. This can be seen in figure 3.

After processing input 1, we reach $B$, as can be seen in figure 4. Right after that we process the next input 0 and switch to state $E$ as shows in figure {fig:second-bit}. The third bit in the string is another 0, which causes us to switch to state $C$ as in figure 6.

What follows is input 1 which makes the automaton switch to state $D$ as in figure 7. Finally, we process another 1 which causes the automaton to switch to state $B$ as in figure 8, which is **not** and accept state and, therefore, the output of the computation is *REJECT*.

The next input is 0, which causes to transition from $B$ to $E$:

One important detail to keep in mind that we don't *ACCEPT* when passing through an accept state, only when the input **ends** at an accept state.

## 7.5 4.202 Language of the automata

Using the same example automaton as before, we can see that that to fall on accept state $E$, the input must end with 0 (see figure 9). $E$ has two entry points which are $B$ and $D$.

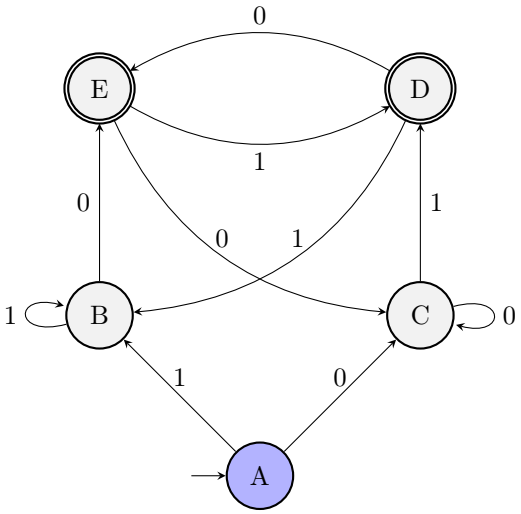



Figure 3: Automaton with 2 inputs: 10011

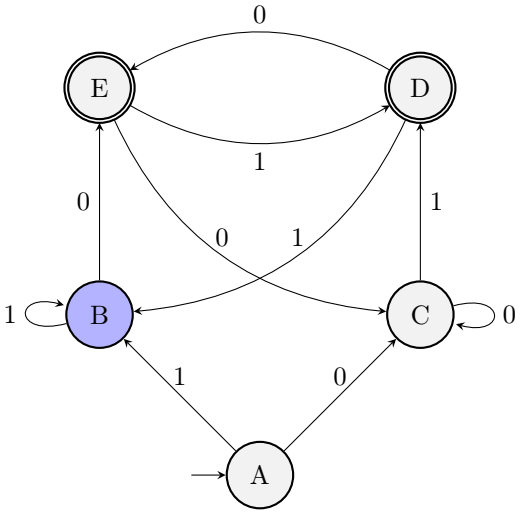



Figure 4: Automaton with 2 inputs: **1**0011

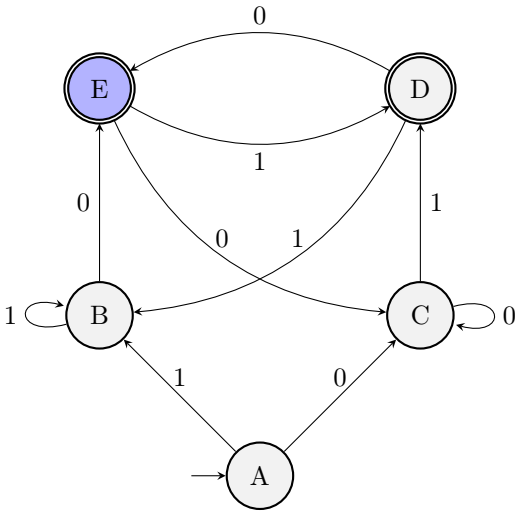



Figure 5: Automaton with 2 inputs: 1**0**011

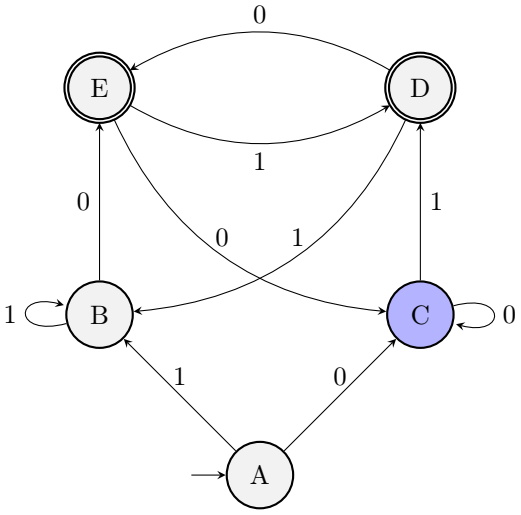



Figure 6: Automaton with 2 inputs: 10**0**11

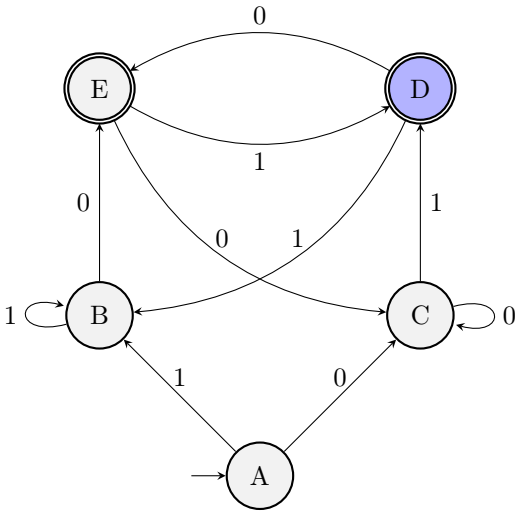



Figure 7: Automaton with 2 inputs: 100**1**1

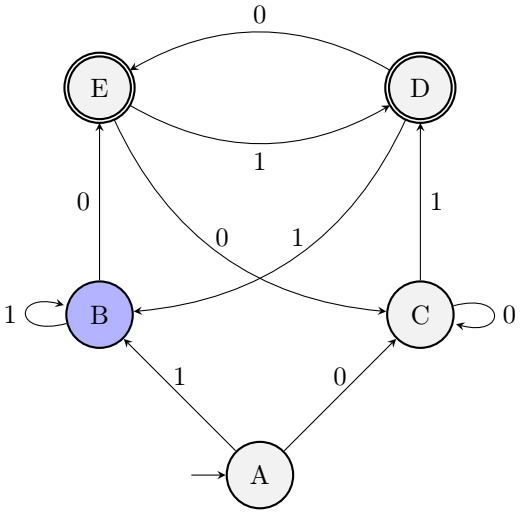



Figure 8: Automaton with 2 inputs: 1001**1**

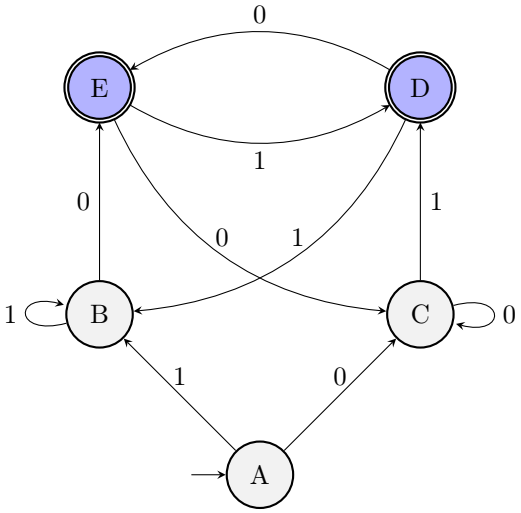



Figure 9: Automaton with 2 inputs: 1001**1**

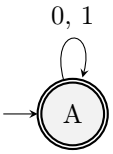



Figure 10: Accepting all binary strings

By looking at the penultimate states, we can see that inputs terminating at the accept state $E$ **must** end with 10.

The only remaining accept state $D$ is symmetrical to $C$ and, therefore, inputs terminating at the accept state $D$ **must** end with 01.

This means that **any** string ending with 01 or 10 will be accepted by our sample automaton.

The set of **all** strings accepted by an automaton is called the **Language** of an automaton. If $M$ is an automaton on alphabet $\Sigma$, then $\mathcal{L}(M)$ is the language of $M$:

$$\mathcal{L}(M) = \{x \in \Sigma^* \mid M \text{ accept } x\}$$

## 7.6 4.204 Recognise a language

Given a set of inputs, can we build an automaton that represents the set of inputs?

### 7.6.1 Accepting all binary strings

The automaton depicted in figure 10 accepts all binary strings regardless of their length.

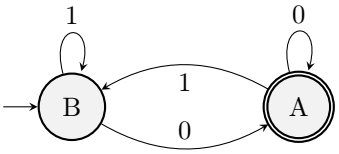



Figure 11: Accepting all binary strings ending with 0

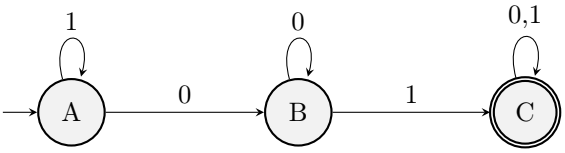



Figure 12: Accepting strings containing with 01

### 7.6.2 Accepting all binary strings ending with 0

The automaton depicted in figure 11 accepts all binary strings ending with 0.

### 7.6.3 Accepting strings ending 01

The automaton depicted in figure 12 accepts all binary strings containing with 01.

### 7.6.4 Automaton accepting strings with 00 or 11

This was left as an exercise. My solution is depicted in figure 13.

# 8 Week 8

Learning Objectives

- Understand the basic terminologies of automata theory.

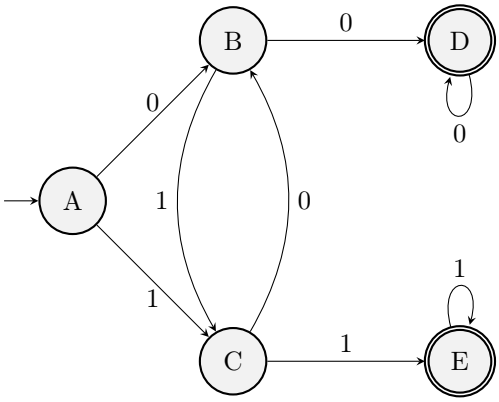



Figure 13: Accepting strings containing 00 or 01

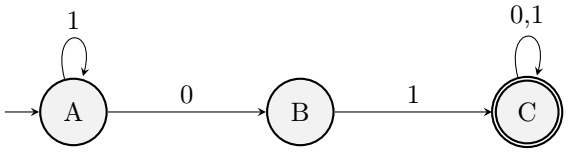



Figure 14: Getting stuck: Not enough Transitions

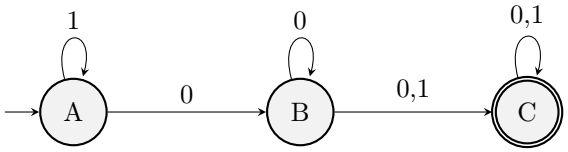



Figure 15: Getting stuck: Too Many Transitions

- Describe finite automata and what it can represent.
- Build deterministic and nondeterministic finite automata.
- Understand and apply various concepts in automata theory, such as deterministic automata, regular languages and context-free grammar.

## 8.1 4.301 Deterministic finite automata (DFA) vs nondeterministic finite automata (NFA)

Sometimes we can get stuck during computation. Figure 14 shows one such example where we can get stuck depending on the input.

For example, if input is *1100*, we can see that we will start initial state $A$ and loop from $A$ to $A$ after the first input *1*. Then the next *1* will cause us to loop again and remain at state $A$.

Then we take a *0* which causes us to move to state $B$. Now, when we take the next *0*, there are no outgoing transitions from $B$ to anywhere else with label *0*, so we get stuck. In the case of figure 14 we get stuck due to a lack of transitions.

Figure 15 has a slightly different problem. Let's work through the states with input *11001*. Again we start at $A$ and take input *1* which causes us to loop back to $A$. Then read another *1* which, again, loops us back to $A$. The next input is *0*, which makes us transition to state $B$. With the next *0* input we have a problem: there are **two** transitions we can take, so which one do we take? We can't make a decision, so we get stuck again. In this case, we get stuck because there are **too many** transitions.

### 8.1.1 Deterministic Finite Automata (DFA)

A DFA is the simplest form of Automata. They are *very well behaved* meaning that we jump from state to state deterministically. What this means is that each state has **exactly** one transition for each character of the alphabet and there is a unique start state.

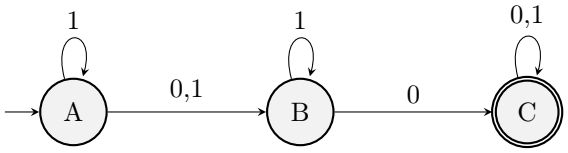



Figure 16: NFA Example

If any of those two requirements are **not** met, then we say the automaton is **non-deterministic**.

### 8.1.2 Non-deterministic Finite Automata (NFA)

A NFA is just a DFA that breaks at least one of the two DFA requirements. For example, we may encounter a state where, for a given input symbol, we can take one of many alternative paths, or no paths at all.

In the context of NFAs, an input is accepted if there is **at least** one sequence of choices that would lead to an accept state.

Because of all these details, the behavior of an NFA can be more complez than DFA.

NFAs can be used in the implementatio of Regular Expressions.

Figure 16 shows an example NFA. Assuming we have the input *1101*, let's work through the computation. When we read the first symbol *1* we already have two choices: loop back $A$ or transition to $B$. Assume we take the loop back to $A$. Another *1* and assume we take the loop again, so we're back at $A$. The following symbol is *0*, so we must transition to $B$. Next, we get a symbol *1* and we loop back to $B$. Because $B$ is not an accept state, the input is rejected.

However, let's try to take another path. With the first symbol *1* we take the transition to $B$. With the next *1*, we loop back to $B$. With the following *0* we transition to $C$ and with the final *1* we loop back to $C$ which is an accept state.

## 8.2 4.303 Computation by NFA

What happens with NFAs that have too few transitions? Using figure 17 as an example, let's work the computation of input *001101*.

1. Start at $A$
2. Read input *0*, transition to $B$
3. Read input *0*, no transition left. We reject input.

### 8.2.1 Language of NFA

Again, we're going to use 17 as our example. We would like to study different inputs.

If input starts with *1*, then we transition to accept state $C$ and anything that follows keeps us in state $C$ by looping back.

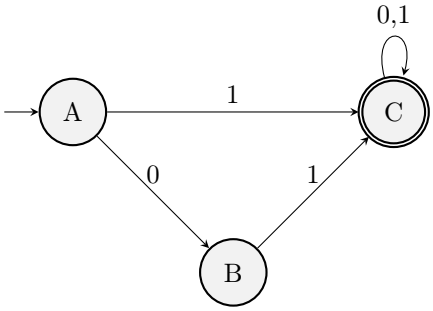



Figure 17: NFA with too few transitions

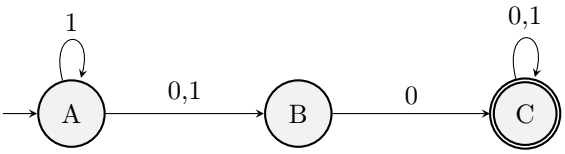



Figure 18: NFA Complex Example

If the input starts with *01*, then we, again, transtion to accept state $C$ (through $B$) and anything that follows keeps us in state $C$ by looping back.

Any other strings will cause us to get stuck.

The language of this NFA is, therefore, all binary strings starting with *1* or *01*.

### 8.2.2 Language of NFA - a complex example

Using figure 18, let's study its language.

If input starts with *0* we transition to $B$. If another *0* follows we transition to accept state $C$. Anything that follows is accepted.

If input starts with *1*, we can either transition $B$ or loop back to $A$. Assuming we loop back to $A$, any following symbol causes us to transition to $B$ which must read a *0* input to transition to accept state $C$. Anything that follows is accepted.

The language of this NFA is, therefore, all binary strings starting with *00* or at least one *1* followed by *0*.

# 9 Week 9

Learning Objectives

- Describe formal languages in the context of regular expressions.
- Identify examples of regular expressions and finite automaton.
- Write regular expressions with and without the use of finite automaton.

## 9.1 5.101 Regular expressions

Before moving on, we review some basic notions and definitions that we will need:

- an **alphabet** $\Sigma$ is a non-empty set of *symbols*
- a **string** or word is a finite sequence of symbols drawn from an alphabet
- empty strings are denoted by $\varepsilon$
- the set of all strings composed from symbols in $\Sigma$ is denoted by $\Sigma^*$
- a language is a collection of strings over an alphabet

Regular expressions are designed around *Regular Operations*. There are three operations at our disposal:

- **Union** $\cup$

  $L_1 \cup L_2 = \{x \mid x \in L_1 \vee x \in L_2\}$
- **Concatenation** $\circ$

  $L_1 \circ L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}$
- **Star** $*$

  $L_1^* = \{x_1 x_2 \dots x_m \mid m \geq 0, x_i \in L_1\}$

### 9.1.1 Example

$$
\begin{aligned}
A &= \{red, green, pink\} \\
B &= \{apple, banana, kiwi\} \\
A \cup B &= \{red, green, pink, apple, banana, kiwi\} \\
A \circ B &= \{redapple, redbanana, redkiwi, greenapple, greenbanana, \\
&\quad greenkiwi, pinkapple, pinkbanana, pinkkiwi\} \\
A^* &= \{\varepsilon, red, green, pink, redred, redgreen, \\
&\quad redpink, greenred, greengreen, greenpink, \\
&\quad pinkred, pinkgreen, pinkpink, \dots\}
\end{aligned}
$$

### 9.1.2 Properties of regular operations

| Operation | Property | Example |
|---|---|---|
| **Union** | Commutative | $A \cup B = B \cup A$ |
| | Associative | $(A \cup B) \cup C = A \cup (B \cup C)$ |
| | Identity | $A \cup \emptyset = A$ |
| | Idempotence | $A \cup A = A$ |
| | Distributive | $(A \cup B) \circ C = (A \circ B) \cup (B \circ C)$ |
| **Concatenation** | Associative | $(A \circ B) \circ C = A \circ (B \circ C)$ |
| | Identity | $A \circ \varepsilon = A$ |
| | Identity | $A \circ \emptyset = \emptyset$ |
| | Distributive | $A \circ (B \cup C) = (A \circ B) \cup (A \circ C)$ |
| **Kleene Star** | | $\emptyset^* = \emptyset$ |
| | | $\varepsilon^* = \varepsilon$ |
| | | $(A^*)^* = A^*$ |
| | | $A^*A^* = A^*$ |
| | | $(A \cup B)^* = (A^*B^*)^*$ |

### 9.1.3 Atomic regular expressions

The emptylanguage $\emptyset$ is a regular expression, which is the empty regular language.

Any symbol $a$ from $\Sigma$ is a regular expression and its regular language is $\{a\}$.

The empty string $\varepsilon$ is a regular expression and its regular language is $\{\varepsilon\}$.

### 9.1.4 Compound regular expressions

By using the regular operations combined with the atomic regular expressions, we can build up compound regular expressions. The operations preserve the regularity, which means that:

- **Concatenation**

  If $\mathrm{R}_1$ and $\mathrm{R}_2$ are regular expressions, so is $R_1 \circ R_2$.

- **Union**

  If $\mathrm{R}_1$ and $\mathrm{R}_2$ are regular expressions, so is $R_1 \cup R_2$.

- **Kleene Star**

  If R is a regular expression, so is $R^*$.

### 9.1.5 The language of the regular expression

- What is the language of $ab^*$?
  - $\{a, ab, abb, abbb, \ldots\}$
- What is the language of $ab^* \cup b^*$?
  - $\{a, ab, abb, abbb, \ldots\} \cup \{\varepsilon, b, bb, bbb, \ldots\} = \{\varepsilon, a, b, ab, bb, abb, bbb, \ldots\}$
- What is the language of $ab^+ \cup b^+b$?

- $\{ab, abb, abbb, \ldots\} \cup \{bb, bbb, bbbb, \ldots\} = ab^* \cup b^* / \{a, \varepsilon, b\}$

### 9.1.6 Examples on binary alphabet $\Sigma = \{a, b\}$

- What is the language of $\Sigma^* a$?
  - $\{a, aa, ba, aaa, aba, \ldots\}$
  - All strings ending with $a$
- What is the language of $\Sigma^* a \Sigma^*$?
  - $\a, aa, ab, ba, aaa, aab, aba, abb, baa, bab, bba, \ldots \
  - All strings containing at least one $a$.

### 9.1.7 Read regular expressions

- Order of Precedence: $*, \circ, \cup$
- Example: Which is the language of $a \cup bc^*$?
  1. $bcbc$
  2. $accc$
  3. $aaa$
  4. $bccc$

  The answer is **4**: $a \cup bc^* = a \cup b(c^*) = a \cup (b(c^*))$. The language is $\{a, b, bc, bcc, bccc, \ldots\}$.

## 9.2 5.103 Design regular expressions

Now we know all the necessary tools to start designing regular expressions for a given language.

### 9.2.1 All binary words containing $bb$

The language of all words containing $bb$ is $\{bb, abb, bba, bbb, aabb, abba, abbb, \ldots\}$. We **know** the word can contain anything before and after $bb$, including the empty string $\varepsilon$. Therefore the regular expression is given by $(a \cup b)^* bb (a \cup b)^*$, which is equivalent to $\Sigma^* bb \Sigma^*$.

### 9.2.2 All binary words ending with $ab$ or $ba$

The language is given by $\{ab, ba, aab, aba, bab, bba, aaab, aaba, \ldots\}$. We can start the word with anything, including $\varepsilon$ as long as it ends with either $ab$ or $ba$.

Therefore, the regular expression is given by $((a \cup b)^* ab) \cup ((a \cup b)^* ba)$ which is equivalent to $(\Sigma^* ab) \cup (\Sigma^* ba)$.

### 9.2.3 All binary words with at most one $a$

The language is given by $\{\varepsilon a, b, ab, ba, bb, abb, bba, bab, \ldots\}$.

The language can contain any number of $b$ before and after one or zero $a$.

Therefore, the regular expression is given by $(b^*ab^*) \cup b^*$.

### 9.2.4 All binary strings of length 3

The language is given by $\aaa, aab, aba, abb, baa, bab, bba, bbb\. This can be expressed by the regular expression $\Sigma\Sigma\Sigma$.

### 9.2.5 All binary strings of length at least 3

This is the same as previous example followed by empty string or any string.

Therefore, $\Sigma\Sigma\Sigma\Sigma^* \equiv \Sigma\Sigma\Sigma^+$

### 9.2.6 All binary strings of length at most 3

This contains:

- All binary strings of length 0: $\varepsilon$
- All binary strings of length 1: $\Sigma$
- All binary strings of length 2: $\Sigma\Sigma$
- All binary strings of length 3: $\Sigma\Sigma\Sigma$

Now we take the union of it all:

$\varepsilon \cup \Sigma \cup \Sigma\Sigma \cup \Sigma\Sigma\Sigma$

## 9.3 5.201 Regular expressions and finite automata

Kleene's Theorem states that a language is regular **if and only if** it can be described by a regular expression.

The application of **if and only if** means we need a two-way theorem:

1. If a language is described by a regular expression, then it is regular.
2. If a language is regular, then it can be described by a regular expression.

We know that the language of Finite Automata is regular and for every regular language there is a Finite Automaton admitting this language. We also know, from Kleene's Theorem, that a regular language can be expressed by a regular expression.

We can use this notion to establish a *link* between regular expressions and finite automata. Also, we can modify Kleene's Theorem in the following way:

1. If $L = L(A)$ for some finite automaton $A$, then there is a regular expression $R$, such that $L(R) = L$.
2. If $L = L(R)$ for some expression $R$, then there is a finite automaton $A$ such that $L(A) = L$.

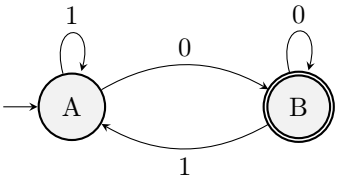



Figure 19: Finite Automaton to Be Converted to Regular Expression

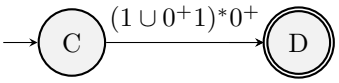



Figure 20: Finite Automaton Converted to Regular Expression

### 9.3.1 Converting Finite Automaton to Regular Expression

We start with a simple two-state Finite Automaton as in figure 19. The first step is to add a new initial state $C$ which a transition $\varepsilon$ from $C$ to $A$.

Then we create a new final state, $D$, with a transition $\varepsilon$ from $B$ to $D$. Our old states, $A$ and $B$, became regular states.

From this point on, we start removing states and transitions by making them more complex. For example, we remove state $B$ by noticing that the path $ABA$ is equivalent to $00^*1$ and the path $ABD$ is equivalent to $00^*$. Therefore, we can remove $B$ and modify the transitions accordingly. When we have multiple transitions from one state to another, we can take the union of all transitions and collapse into a single transition.

Also note that $00^*$ is the same as $0^+$, so we can can simplify the transition from $A$ to $D$. Finally, we can remove state $A$ by realising that we have a single path from $C$ to $D$ equivalent to $(1 \cup 0^+1)^*0^+$. The result is shown in figure 20.

## 9.4 5.203 Regular expressions and finite automata: examples

The following figures 21, 22, 23, 24, 25, and 26 show the entire process of converting a finite automaton to regular expression.

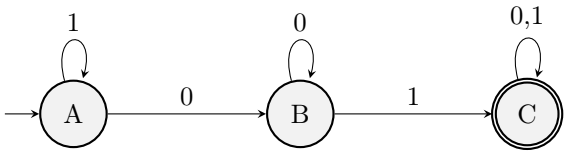



Figure 21: Step 0: Initial Finite Automaton

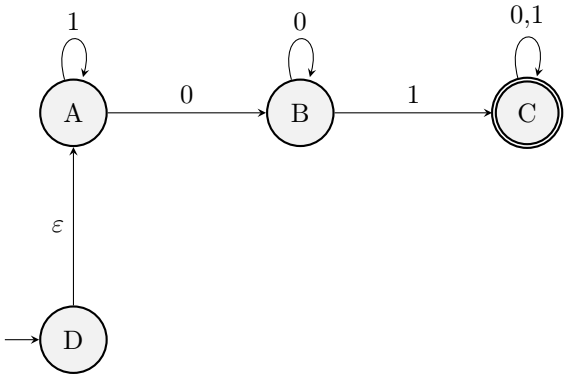



Figure 22: Step 1: Add new initial state

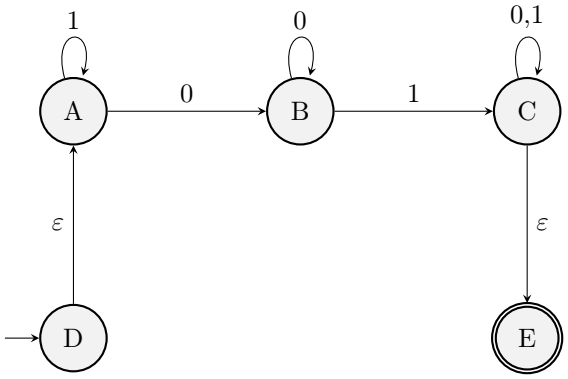



Figure 23: Step 2: Add new final state

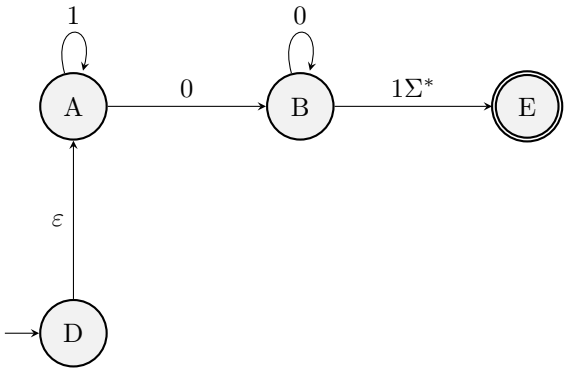



Figure 24: Step 3: Remove state C

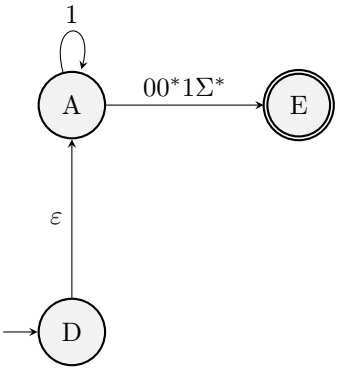



Figure 25: Step 4: Remove state B

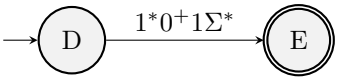



Figure 26: Step 5: Remove state A

# 10 Week 10

Learning Objectives

- Describe formal languages in the context of regular expressions
- Identify examples of regular expressions and finite automaton.
- Write regular expressions with and without the use of finite automaton.

## 10.1 5.301 Regular or non-regular?

A language is referred to as *regular* if it can be accepted by a finite automaton. Moreover, regular languages can be accepted by regular expressions. Every finite language is regular.

The problem here is that building finite automatons and regular expressions is a non-obvious task; therefore we must implement other techniques such as breaking a language into smaller regular languages and building the language back up by relying on properties of regular languages.

### 10.1.1 Closure properties

**Theorem 15.** *If $L_1$ and $L_2$ are regular languages on alphabet $\Sigma$, then the following languages are also regular:*

- *$U - L_1$: This means $\Sigma^* - L_1$ or the complement of $L_1$*
- *$L_1 \cup L_2$: The union of $L_1$ and $L_2$*
- *$L_1 \cap L_2$: The intersection of $L_1$ and $L_2$*
- *$L_1L_2$: The product of $L_1$ and $L_2$*

- $L_1{}^*$: *The Kleene star of* $L_1$

### 10.1.2 How can we show a language is non-regular?

A non-regular language **cannot** be accepted by a finite automaton, however we cannot test **all** finite automata. Likewise, we cannot test all regular expressions.

### 10.1.3 Example of non-regular languages

- $L = \{a^n b^n \mid n \in \mathbb{N}\}$

  Any number of $a$ followed by any number of $b$

- $L = \{xx \mid x \in \{a, b\}^*\}$

  The language of all binary strings concatenated with itself

- $L = \{xx^R \mid x \in \{a, b\}^*\}$

  The language of all binary strings concatenated with its reverse

- $L = \{a^{n!} \mid n \in \mathbb{N}\}$

  The language composed of all strings in the form of n-factorial numbers of $a$

- $L = \{a^{n^2} \mid n \in \mathbb{N}\}$

  All the strings in the form of perfect square number of $a$

- $L = \{a^n \mid n \in \mathbb{N}$, n is a prime number$\}$

  All the strings in the form of prime number of $a$

### 10.1.4 Using closure properties - intersection

One powerful technique to show that a language is non-regular is by employing closure properties.

Prove $L = \{x \in \{a, b\}^* \mid \#\text{a in x} = \#\text{b in x}\}$ is not regular. Let's list a few strings in this language:

$$L = \{ab, aabb, abab, abba, baab, \ldots\}$$

*Proof.* By contradiction.

Let's assume $L$ to be regular. We know that $L' = \{x \in a^*b^*\}$ is regular. We also know that the intersection of two regular languages is also regular, therefore $L \cap L' = \{a^n b^n \mid n \in \mathbb{N}\}$ must be regular. We know this language to not be regular, therefore L cannot be a regular language. □

### 10.1.5 Using closure properties - complement

Prove $L = \{a^i b^j \mid i, j \in \mathbb{N}, i \neq j\}$ is not regular.

Looking at a few example strings from this language:

$$L = \{abb, aab, abbb, aaabb, \ldots\}$$

*Proof.* By contradiction.

Let's assume $L$ to be regular, therefore $\neg L$ must also be regular.

We know $\neg L = \{a^n b^n\} \cup \text{non-bitonic}$ is regular. We also know $L' = \{x \in a^* b^*\}$ is regular. Moreover, we know that the intersection of two regular languages is also regular, therefore $\neg L \cap L'$ must be regular.

However, $\neg L \cap L' = \{a^n b^n \mid n \in \mathbb{N}\}$ is non-regular. □

## 10.2 5.303 Pumping lemma

The Pumping Lemma explains a key property of regular languages. The essence of the pumping lemma is about finding a loop or repeated substrings.