Lecture 2, Part 1: Boolean Algebra

August 27, 2026

Lecturer and scribe: Karthik Gajulapalli

---

1 Last Time and Today

Last class, we saw the famous stable matching algorithm and discussed a high-level sketch of how to prove complex statements. In particular, we considered how to prove that the algorithm we developed in class actually outputs a stable matching.

In today's lecture, we will focus on two parts:

1. We will give a quick survey of Boolean algebra, which will come up throughout the class.
2. We will prove from first principles that $1 + 2 = 3$. (See Part 2.)

2 Boolean Algebra

In high school, you probably encountered several different number systems. For example, in calculus, many of the quantities you work with are **Real numbers** ($\mathbb{R}$). In geometry or algebra, you may have worked with **Integers** ($\mathbb{Z}$) or **Natural Numbers** ($\mathbb{N}$).

Here are some examples:

- **Natural Numbers** ($\mathbb{N}$): $0, 1, 2, 3, 4, \ldots$ — often used when counting objects or describing the size of a set.
- **Integers** ($\mathbb{Z}$): $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$ — useful for quantities that can be positive or negative, such as temperature or changes in elevation.
- **Rational Numbers** ($\mathbb{Q}$): numbers that can be written as a fraction $\frac{p}{q}$, where $p, q \in \mathbb{Z}$ and $q \neq 0$. For example, $\frac{1}{2}$, $-\frac{7}{3}$, and 5.
- **Real Numbers** ($\mathbb{R}$): all numbers that can be represented on the number line, including both rational and irrational numbers such as $\sqrt{2}$ and π.
- **Complex Numbers** ($\mathbb{C}$): numbers of the form $a + bi$, where $a, b \in \mathbb{R}$ and $i^2 = -1$. These often appear when solving equations such as $x^2 + 1 = 0$.

These number systems are not completely separate from one another. Instead, they form a hierarchy:

$$\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}.$$

As we move from left to right, we allow ourselves a larger collection of numbers, which lets us solve more types of equations and describe more mathematical objects.

While each of the number systems above contains infinitely many possible values, in today's lecture we will explore what happens when we restrict ourselves to just two values: 0 and 1.

It may seem like an incredibly small collection, but it turns out that working with just these two values is remarkably powerful. In fact, these two values form the foundation of modern computing: everything from the programs we write to the images, videos, and websites we use can ultimately be represented using just 0's and 1's.

2.1 Boolean Values and Functions

We will use the values 0 and 1, and the values **False** and **True**, interchangeably. In other words, we can think of 0 as False and 1 as True.

A **Boolean function** is a function whose inputs and outputs are restricted to $\{0, 1\}$. For example, a Boolean function of two variables has the form

$$f : \{0, 1\}^2 \to \{0, 1\}.$$

When working with variables that can take only the values 0 or 1, we need to define our operations carefully. For example, if we add $1 + 1$, then in the natural numbers we would normally get 2. However, 2 is not one of the values allowed in our Boolean system. Thus, ordinary addition is not an operation on $\{0, 1\}$.

We therefore need to define operations that always take values in $\{0, 1\}$. These operations form the foundation of Boolean algebra, which is the mathematical language underlying much of modern computing.

2.2 Truth Tables

A Boolean function can be completely described by a **truth table**, which lists the output of the function for every possible input.

For two variables x and y, there are four possible inputs:

$$(0, 0), \quad (0, 1), \quad (1, 0), \quad (1, 1).$$

Here are some examples of Boolean functions on two variables:

x	y	0
0	0	0
0	1	0
1	0	0
1	1	0

Constant function

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

OR function

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

AND function

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

XOR function

We call the tables above **truth tables** for their respective functions. Each truth table completely specifies one Boolean function.

Since there are four possible inputs (x, y) and the output for each input can independently be either 0 or 1, there are

$$2^4 = 16$$

possible Boolean functions on two variables.[1]

2.3 Boolean Operations

We will use the notation $\neg x$ to denote the negation of x. In other words,

$$\neg x = \begin{cases} 0 & \text{if } x = 1, \\ 1 & \text{if } x = 0. \end{cases}$$

Its truth table is

x	$\neg x$
0	1
1	0

We will primarily work with the three operations

$$\neg, \qquad \wedge, \qquad \vee,$$

which we refer to as **NOT**, **AND**, and **OR**.

[1] We will do more counting later in class. For now, the easiest way to think about this is that we have four slots, one for each possible input, and each slot can contain either a 0 or a 1.

2.4 Some Boolean Laws

The following identities are useful when manipulating Boolean expressions.

$x \vee 0 = x$	$x \wedge 1 = x$	(Identity)
$x \vee 1 = 1$	$x \wedge 0 = 0$	(Domination)
$x \vee x = x$	$x \wedge x = x$	(Idempotent)
$x \vee \neg x = 1$	$x \wedge \neg x = 0$	(Complement)
$\neg(\neg x) = x$		(Double Negation)
$x \vee y = y \vee x$	$x \wedge y = y \wedge x$	(Commutative)
$(x \vee y) \vee z = x \vee (y \vee z)$	$(x \wedge y) \wedge z = x \wedge (y \wedge z)$	(Associative)
$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$		(Distributive)
$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$		(Distributive)
$\neg(x \vee y) = \neg x \wedge \neg y$	$\neg(x \wedge y) = \neg x \vee \neg y$	(De Morgan's)

Example: Verifying De Morgan's Law. We can prove a Boolean identity by checking that both sides have the same value on every input. For example, consider

$$\neg(x \vee y) = \neg x \wedge \neg y.$$

The following truth table includes the intermediate calculations:

x	y	$x \vee y$	$\neg(x \vee y)$	$\neg x$	$\neg y$	$\neg x \wedge \neg y$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

The columns for $\neg(x \vee y)$ and $\neg x \wedge \neg y$ agree in every row. Since these are all possible inputs, the identity holds for every choice of Boolean values x and y.

2.4.1 Logical Equivalence

Two Boolean expressions A and B are **logically equivalent** if they have the same truth value for every assignment of values to their variables. We write

$$A \equiv B.$$

In a truth table, this means that their output columns agree in every row, including rows where both expressions are false. Equivalently, the two expressions describe the same Boolean function.

Our verification of De Morgan's law therefore shows that

$$\neg(x \vee y) \equiv \neg x \wedge \neg y.$$

The Boolean identities listed above all assert logical equivalences: their equalities hold for every input.

Agreement on a single input is not enough. For example, $x \vee y$ and $x \wedge y$ are both 1 when $x = y = 1$, but when $x = 1$ and $y = 0$ their values are 1 and 0, respectively. Thus they are not logically equivalent. One input on which two expressions differ is enough to disprove logical equivalence.

Logical equivalence lets us replace an expression inside a larger Boolean expression without changing its value. For example,

$$\neg(x \vee y) \vee z \equiv (\neg x \wedge \neg y) \vee z.$$

This is why Boolean laws are useful when simplifying expressions.

2.5 Disjunctive Normal Form

A useful way to represent Boolean functions is through their *disjunctive normal form* (DNF).

A Boolean expression is in **disjunctive normal form** if it is an OR of ANDs of literals, where a *literal* is either a variable or its negation.

For example,

$$(x \wedge \neg y) \vee (\neg x \wedge z) \vee (x \wedge y \wedge z)$$

is in disjunctive normal form.

Each expression inside a parenthesis is called a *term*.

2.5.1 Every Boolean Function Can Be Written in DNF

For two variables x and y, there are only four possible inputs:

$$(0,0), \quad (0,1), \quad (1,0), \quad (1,1).$$

The key idea is that we can use an AND expression to "pick out" any one of these inputs:

$$\begin{aligned}(0,0) &\longrightarrow \neg x \wedge \neg y,\\ (0,1) &\longrightarrow \neg x \wedge y,\\ (1,0) &\longrightarrow x \wedge \neg y,\\ (1,1) &\longrightarrow x \wedge y.\end{aligned}$$

Each expression is 1 on exactly one input.

Therefore, to represent any Boolean function, simply look at the rows where the function is 1 and OR together the corresponding expressions.

For example, return to the XOR function from our earlier truth tables. Writing $f(x,y) = x \oplus y$, its truth table is

x	y	$f(x,y)$
0	0	0
0	1	1
1	0	1
1	1	0

The function is 1 on the inputs $(0,1)$ and $(1,0)$. Therefore,

$$f(x,y) = x \oplus y = (\neg x \wedge y) \vee (x \wedge \neg y).$$

Thus,

AND picks out rows, and OR combines the rows where $f = 1$.

It is not hard to see how this strategy generalizes even when dealing with more than two variables.

2.6 Functional Completeness

We can go one step further. Notice that DNF expressions use three operations:

$$\neg, \qquad \wedge, \qquad \vee.$$

However, the OR operation can itself be written using only NOT and AND. By De Morgan's Law,

$$x \vee y = \neg(\neg x \wedge \neg y).$$

Therefore, every occurrence of $\vee$ in a DNF expression can be replaced by

$$A \vee B = \neg(\neg A \wedge \neg B).$$

For example,

$$(x \wedge \neg y) \vee (\neg x \wedge z)$$

can be rewritten as

$$\neg\left(\neg(x \wedge \neg y) \wedge \neg(\neg x \wedge z)\right).$$

This expression uses only the operations $\neg$ and $\wedge$.

Since every Boolean function can be written in DNF, and every DNF expression can be rewritten using only NOT and AND, we obtain the following result:

Every Boolean function can be computed using only NOT and AND gates.

Thus, surprisingly, we do not need separate AND, OR, and NOT gates to compute every Boolean function. NOT and AND alone are enough.

We say that the set

$$\boxed{\{\neg, \wedge\}}$$

is *functionally complete*.

Expressibility and efficiency. Functional completeness tells us that an expression exists for every Boolean function; it does not guarantee a short expression. A truth table on n inputs has 2^n rows, so our DNF construction may produce as many as 2^n terms, each containing n literals. An equivalent expression can sometimes be much shorter. Finding a compact representation is a separate question from showing that a representation exists. It turns out that finding the most compact represenation of a function is one of the most central problems in computer science and widely believed to be a very hard task.