Fundamental Algorithm Techniques
Problem Set #9

Yessengaliyeva Gulnazym

December 2025

Problem 1: Finite Functions on the Computer

We consider finite functions defined on binary vectors:

$$F : \{0,1\}^n \longrightarrow Y,$$

where $\{0,1\}^n$ is the set of all binary strings of length n. The size of the input space is

$$|\{0,1\}^n| = 2^n.$$

A function F is completely determined by the values it takes on each of these 2^n inputs.

In each of the following cases we want to count the number of possible functions F:

(a) $Y = \{0,1\}$,

(b) $Y = \{-1,0,1\}$,

(c) $Y = \{0,1\}^m$.

It is also helpful to view F via a decision tree on n binary variables, with 2^n leaves labelled by the output values.

(a) Output $Y = \{0,1\}$

In this case, for each input $x \in \{0,1\}^n$ the function value $F(x)$ can be either 0 or 1.

There are 2^n different inputs, and for each of them we have 2 choices for the value of F. Hence, the total number of possible functions is

$$\underbrace{2 \times 2 \times \cdots \times 2}_{2^n \text{ times}} = 2^{2^n}.$$

Conclusion. The number of functions

$$F : \{0,1\}^n \longrightarrow \{0,1\}$$

is

$$\boxed{2^{2^n}}.$$

(b) Output $Y = \{-1, 0, 1\}$

Now the output set consists of 3 different values. For each input $x \in \{0,1\}^n$ we can choose $F(x) \in \{-1, 0, 1\}$.

Again there are 2^n possible inputs, and for each input we have 3 options. Thus the number of different functions is

$$\underbrace{3 \times 3 \times \cdots \times 3}_{2^n \text{ times}} = 3^{2^n}.$$

Conclusion. The number of functions

$$F : \{0,1\}^n \longrightarrow \{-1, 0, 1\}$$

is

$$\boxed{3^{2^n}}.$$

(c) Output $Y = \{0,1\}^m$

The set $\{0,1\}^m$ is the set of all binary vectors of length m, so it has

$$\left|\{0,1\}^m\right| = 2^m$$

elements. For each input $x \in \{0,1\}^n$ we can choose any of these 2^m possible outputs.

Since there are 2^n different inputs and 2^m choices for each input, the total number of functions is

$$\underbrace{2^m \times 2^m \times \cdots \times 2^m}_{2^n \text{ times}} = (2^m)^{2^n} = 2^{m \cdot 2^n}.$$

Conclusion. The number of functions

$$F : \{0,1\}^n \longrightarrow \{0,1\}^m$$

is

$$\boxed{2^{m \cdot 2^n}}.$$

Decision tree interpretation

A convenient way to visualise such functions is via a decision tree. We consider a full binary tree of depth n, where each level corresponds to one input bit. The root queries x_1, the next level queries x_2, and so on, until the n-th level queries x_n. Thus, the tree has 2^n leaves, each leaf corresponding to a unique input $x \in \{0,1\}^n$.

Each leaf is labelled with the output value $F(x) \in Y$. Therefore, the number of different functions is equal to the number of ways to label the 2^n leaves by elements of Y. If $|Y| = k$, then

$$\#\text{functions} = k^{2^n},$$

which recovers the formulas in parts (a), (b), and (c).

Below we illustrate the decision tree for the case $n = 2$.

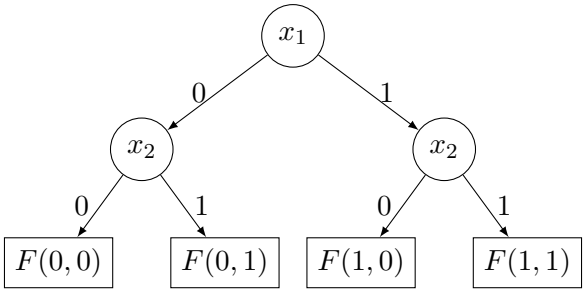

Figure 1: Decision tree for a function $F : \{0,1\}^2 \to Y$.

Problem 2: Equivalence NAND $\Rightarrow$ AND, OR & NOT

We consider the Boolean operation NAND, denoted by $A \uparrow B$, with the following truth table:

A	B	$A \uparrow B$
0	0	1
0	1	1
1	0	1
1	1	0

By definition,

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

The goal is to show that the logical operations NOT, AND, and OR can be implemented using only NAND gates. This implies that NAND is a universal Boolean operation.

NOT from NAND

Negation can be obtained by feeding the same input into both inputs of a NAND gate:

$$\neg A = A \uparrow A.$$

Figure 2: NOT implemented using a single NAND gate: $\neg A = A \uparrow A$.

AND from NAND

Since $A \uparrow B = \neg(A \wedge B)$, we can recover AND by negating the output once more:

$$A \wedge B = (A \uparrow B) \uparrow (A \uparrow B).$$

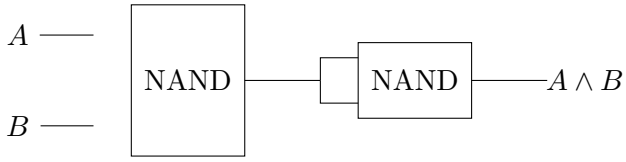

Figure 3: AND implemented using two NAND gates.

OR from NAND

Using De Morgan's law,

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

Since

$$\neg A = A \uparrow A, \quad \neg B = B \uparrow B,$$

we obtain:

$$A \vee B = (A \uparrow A) \uparrow (B \uparrow B).$$

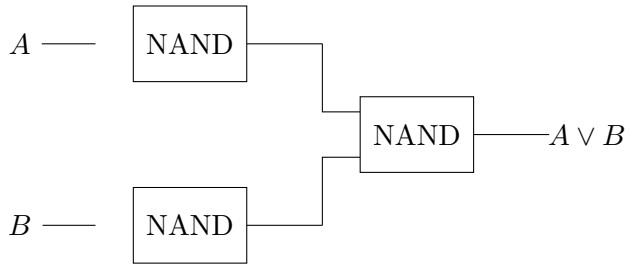

Figure 4: OR implemented using three NAND gates.

Universality of NAND

Any Boolean circuit built from AND, OR, and NOT gates can be transformed into an equivalent circuit using only NAND gates:

- NOT requires one NAND gate;
- AND requires two NAND gates;
- OR requires three NAND gates.

Therefore, if a Boolean circuit has n gates, it can be replaced by an equivalent circuit consisting of at most $3n$ NAND gates. This proves that NAND is a universal Boolean operation.

Problem 3: Universality of Boolean Circuits

Let

$$F : \{0,1\}^n \longrightarrow \{0,1\}$$

be an arbitrary Boolean function.

Indicator functions δ_x

For each input vector $x = (x_1, \ldots, x_n) \in \{0,1\}^n$, we define the function

$$\delta_x(y) = \begin{cases} 1, & \text{if } y = x, \\ 0, & \text{otherwise.} \end{cases}$$

This function takes the value 1 on exactly one input x and 0 on all other inputs.

Circuit size of δ_x

The function δ_x can be implemented by checking bitwise equality between y and x. For each coordinate i, we use y_i if $x_i = 1$ and $\neg y_i$ if $x_i = 0$, and then combine all these conditions using logical AND:

$$\delta_x(y) = \bigwedge_{i=1}^{n} \begin{cases} y_i, & x_i = 1, \\ \neg y_i, & x_i = 0. \end{cases}$$

This construction uses $O(n)$ logical gates, and therefore the circuit size of δ_x is $O(n)$.

Construction of F

Let

$$S = \{x \in \{0,1\}^n \mid F(x) = 1\}$$

be the set of inputs on which F outputs 1. Then the function F can be expressed as

$$F(y) = \bigvee_{x \in S} \delta_x(y).$$

Complexity bound

In the worst case, $|S| \leq 2^n$. Since each function δ_x requires $O(n)$ gates, the total circuit size needed to compute F is

$$O(n \cdot 2^n).$$

Conclusion. Every Boolean function $F : \{0,1\}^n \to \{0,1\}$ is computable by a Boolean circuit of size $O(n \cdot 2^n)$.