4

Syntactic sugar, and computing every function

“The [In 1951] I had a running compiler and nobody would touch it because, they carefully told me, computers could only do arithmetic; they could not do programs.”, Grace Murray Hopper, 1986.

“Syntactic sugar causes cancer of the semicolon.”, Alan Perlis, 1982.

The computational models we considered thus far are as “bare bones” as they come. For example, our NAND-CIRC “programming language” has only the single operation \( \text{foo} = \text{NAND} (\text{bar}, \text{blah}) \). However, in this chapter we will show that these simple models are actually equivalent to more powerful ones. The key observation is that we can implement more complex features using our basic building blocks, and then use these new features themselves as building blocks for even more sophisticated features. This is known as “syntactic sugar” in the field of programming language design, since we are not modifying the underlying programming model itself but rather we merely implement new features by syntactically transforming a program that uses such features into one that doesn’t.\(^1\)

\( ^1 \) This concept is also known as “macros” or “meta-programming” and is sometimes implemented via a preprocessor or macro language in a programming language or a text editor. One modern example is the Babel JavaScript syntax transformer, that converts JavaScript programs written using the latest features into a format that older Browsers can accept. It even has a plug-in architecture, that allows users to add their own syntactic sugar to the language.

\[ \text{Example 4.1 — Constants in NAND-CIRC.} \] The NAND-CIRC programming language is so “bare bones” that it does not even allow for literal constants. That is, although the variables in NAND-CIRC are Boolean and can store either the value 0 or 1, it does not explicitly allow for assigning a constant 0 or 1 into a variable. That is, we cannot write code such as \( \text{foo} = 1 \) or \( \text{bar} = 0 \) in NAND-CIRC. However, we can still achieve the same effect by using the observation that for every \( a \in \{0, 1\} \), the AND of \( a \) and NOT(\( a \)) is always 0, and hence the \( \text{NAND} (a, \text{NAND} (a, a)) = 1 \).

Learning Objectives:
- Get comfort with syntactic sugar or automatic translation of higher level logic to low level gates.
- Learn proof of major result: every finite function can be computed by a Boolean circuit.
- Start thinking \emph{quantitatively} about number of lines required for computation.
Therefore, we can always create variables zero and one that have the value 0 and 1 using the following code:

\[
\begin{align*}
\text{temp} &= \text{NAND}(X[0], X[0]) \\
\text{one} &= \text{NAND}(\text{temp}, X[0]) \\
\text{zero} &= \text{NAND}(\text{one}, \text{one})
\end{align*}
\]

This means that whenever we need to, we can assume we have access to such variables in NAND-CIRC.

This chapter provides a “toolkit” that can be used to show that many functions can be computed by (not too long) NAND-CIRC programs, and hence also by Boolean circuit (with not too many gates). We will also use this toolkit to prove a fundamental theorem: every finite function \( f : \{0, 1\}^n \to \{0, 1\}^m \) can be computed by a Boolean circuit, see Theorem 4.12 below. While the syntactic sugar toolkit is important in its own right, the latter theorem can also be proven directly using Boolean circuits. We present this alternative proof in Section 4.5. See Fig. 4.1 for an outline of the results of this section.

**Figure 4.1**: An outline of the results of this chapter. In Section 4.1 we give a toolkit of “syntactic sugar” transformations showing how to implement features such as programmer-defined functions and conditional statements in NAND-CIRC. We use these tools in Section 4.3 to give a NAND-CIRC program (or alternatively a Boolean circuit) to compute the \textit{LOOKUP} function. We then build on this result to show in Section 4.4 that NAND-CIRC programs (or equivalently, Boolean circuits) can compute every finite function. An alternative direct proof of the same result is given in Section 4.5.

**Remark 4.2** — Syntactic sugar for circuits. In this chapter we focus on the straight-line programming language view of our computational models, and specifically the NAND-CIRC programming language. This is because many of the syntactic sugar transformations we present are easiest to think about in terms of applying “search and replace” operations to the source code of a program. However, by Theorem 3.20, all of our results hold equally well for other models including...
the standard model Boolean circuits that use the AND, OR, NOT operations. In particular, we can use similar transformations to show that augmenting our circuit model with additional gates does not change its computational power.

4.1 SOME EXAMPLES OF SYNTACTIC SUGAR

We now present some examples of “syntactic sugar” that we can use in constructing NAND-CIRC programs. This is not an exhaustive list - if you find yourself needing to use an extra feature in your NAND-CIRC program then you can just show how to implement it based on the existing ones. Going over examples for syntactic sugar can be a little tedious, but we do it for two reasons:

1. To convince you that despite its seeming simplicity and limitations, the NAND-CIRC programming language is actually quite powerful and can capture many of the fancy programming constructs such as if statements and function definitions that exists in more fashionable languages.

2. So you can realize how lucky you are to be taking a theory of computation course and not a compilers course... :)

**Remark 4.3 — Counting lines.** While we can use syntactic sugar to present NAND-CIRC programs in more readable ways, we did not change the definition of the language itself. Therefore, whenever we say that some function $f$ has an $s$-line NAND-CIRC program we mean a standard “sugar free” NAND-CIRC program, where all syntactic sugar has been expanded out. For example, the program above is a 12-line program for computing the MAJ function, even though it can be written in fewer lines using the function definition syntactic sugar.

4.1.1 Functions / Macros

One staple of almost any programming language is the ability to define and then execute functions. However, we can achieve the same effect as (non recursive) functions using the time honored technique of “copy and paste”. That is, we can replace code which defines a macro such as

```python
def Func(a,b):
    function_code
```

\[^2\] This was not always the case. For example, original version of the FORTRAN programming language, developed in the early 1950’s, did not have support for user-defined functions. This was however quickly added in FORTRAN II, released in 1958.
return c
some_code
f = Func(e,d)
some_more_code

with the following code where we “paste” the code of Func

some_code
function_code'
some_more_code

and where function_code' is obtained by replacing all occurrences of a with d, with e, c with f. When doing that we will need to ensure that all other variables appearing in function_code' don’t interfere with other variables. We can always do so by renaming variables to new names that were not used before.

Example 4.4 — Computing Majority from NAND using syntactic sugar. Functions allow us to express NAND-CIRC programs much more cleanly and succinctly. For example, because we can compute AND, OR, and NOT using NANDs, we can compute the Majority function as follows:

def NOT(a):
    return NAND(a,a)
def AND(a,b):
    temp = NAND(a,b)
    return NOT(temp)
def OR(a,b):
    temp1 = NOT(a)
    temp2 = NOT(b)
    return NAND(temp1,temp2)
def MAJ(a,b,c):
    and1 = AND(a,b)
    and2 = AND(a,c)
    and3 = AND(b,c)
    or1 = OR(and1,and2)
    return OR(or1,and3)

print(MAJ(0,1,1)) # 1

4.1.2 Proof by Python (optional)
We can write a program that takes a NAND-CIRC program $P$ that includes function definitions and using simple “search and replace”
transform $P$ into a standard (i.e., “sugar free”) NAND-CIRC program $P'$ that computes the same function as $P$. The idea is simple: if the program $P$ contains a definition of a function $\text{Func}$ of two arguments $x$ and $y$, then whenever we see a line of the form $\text{foo} = \text{Func}(\text{bar}, \text{blah})$, then we replace this line by the body of the function $\text{Func}$ (replacing all occurrences of $x$ and $y$ with $\text{bar}$ and $\text{blah}$ respectively). If the last line of $\text{Func}$ was $\text{return exp}$ where $\text{exp}$ is some expression, then we replace ending with the line $\text{foo} = \text{exp}$ (where $\text{exp}$ is some expression) then we replace it with $\text{foo} = \text{exp}$.3

The following Python code achieves such a transformation:4

```python
import re
def desugar(code, func_name, func_args, func_body):
    """Use 'search and replace' to replace calls to a function with its code. Uses Python regular expressions to simplify the search and replace, see https://docs.python.org/3/library/re.html

If we have a function $\text{FUNC}$ with arguments $x$ and $y$ and where its last line has the form 'return expression' then we will replace every line in our code of the form

$\text{foo} = \text{FUNC}(a,b)$

with

$\text{func_body}[x->a,y->b]$

$\text{foo} = \text{expression}$

"""

    # regular expression for capturing a list of variable names separated by commas
    arglist = ",".join([r"([a-zA-Z0-9_\[\]\]")] for i in range(len(func_args))]
```
# regular expression for capturing a statement of the form
# "variable = func_name(arguments)"
regexp = fr'\([a-zA-Z0-9_]\[\]\]+\s*=\s*{func_name}\(\{arglist\}\)\s*\$'
m = re.search(regexp, code, re.MULTILINE)
if not m: return code # if no match then there's nothing to do
newcode = func_body

# replace function arguments by the variables from the function invocation
for i in range(len(func_args)):
    newcode = newcode.replace(func_args[i], m.group(i+2))

# Splice the new code inside
newcode = newcode.replace('return', m.group(1)+" = ")
newcode = code[:m.start()] + newcode + code[m.end()+1:]
# Continue recursively to check if there are more matches
return desugar(newcode, func_name, func_args, func_body)

Fig. 4.2 shows the result of applying this code to the program of ?? that uses syntactic sugar to compute the Majority function. (Specifically, we first apply desugar to remove usage of the OR function, then apply it to remove usage of the AND function, and finally apply it a third time to remove usage of the NOT function.)

Remark 4.5 — Parsing function definitions. In the code above, we assumed that we are given the function already split up into its name, arguments, and body. It is not crucial for our purposes to describe precisely to scan a definition and splitting it up to these components, but in case you are curious, it can be achieved in Python via the following code:

def parse_func(code):
    '''Parse a function definition into name, arguments and body'''
    lines = [l.strip() for l in code.split('\n')]
    regexp = r'\s+\([a-zA-Z0-9_]\[\]\]+\s*\(\{\[a-zA-Z0-9_,\]\]+\)\s*\:\s*\$'}
syntactic sugar, and computing every function

m = re.match(regexp, lines[0])
return m.group(1), m.group(2).split(',')
  \njoin(lines[1:])

### 4.1.3 Conditional statements

Another sorely missing feature in NAND-CIRC is a conditional statement such as the `if/then` constructs that are found in many programming languages. However, using functions, we can obtain an ersatz `if/then` construct. First we can compute the function $IF: \{0, 1\}^3 \to \{0, 1\}$ such that $IF(a, b, c)$ equals $b$ if $a = 1$ and $c$ if $a = 0$.

Before reading onward, try to see how you could compute the $IF$ function using NAND’s. Once you do that, see how you can use that to emulate `if/then` types of constructs.

The $IF$ function can be implemented from NANDs as follows (see Exercise 4.2):

```python
def IF(cond, a, b):
    notcond = NAND(cond, cond)
    temp = NAND(b, notcond)
    temp1 = NAND(a, cond)
    return NAND(temp, temp1)
```

```python
print(IF(0, 1, 0))  # 0
print(IF(1, 1, 0))  # 1
```

The $IF$ function is also known as the multiplexing function, since $cond$ can be thought of as a switch that controls whether the output is connected to $a$ or $b$.

Using the $IF$ function, we can implement conditionals in NAND. The idea is that we replace code of the form

```python
if (condition): assign blah to variable foo
```

with code of the form

```python
foo = IF(condition, blah, foo)
```

that assigns to $foo$ its old value when $condition$ equals 0, and assign to $foo$ the value of $blah$ otherwise. More generally we can replace code of the form
if (cond):
    a = ...
    b = ...
    c = ...

    with code of the form

    temp_a = ...
    temp_b = ...
    temp_c = ...
    a = IF(cond, temp_a, a)
    b = IF(cond, temp_b, b)
    c = IF(cond, temp_c, c)

4.2 EXTENDED EXAMPLE: ADDITION AND MULTIPLICATION (OPTIONAL)

Using “syntactic sugar”, we can write the integer addition function as follows:\(^5\)

\[
\text{def ADD}(A,B):
\]
\[
\text{Result} = [0]^{n+1}
\]
\[
\text{Carry} = [0]^{n+1}
\]
\[
\text{Carry}[0] = \text{zero}(A[0])
\]
\[
\text{for i in range}(n):
\]
\[
\text{Result}[i] = \text{XOR} (\text{Carry}[i], \text{XOR}(A[i],B[i]))
\]
\[
\text{Carry}[i+1] = \text{MAJ}(\text{Carry}[i], A[i], B[i])
\]
\[
\text{Result}[n] = \text{Carry}[n]
\]
\[
\text{return Result}
\]

ADD([1,1,1,0,0],[1,0,0,0,0]);;
# [0, 0, 0, 1, 0, 0]

where zero is the constant zero function, and MAJ and XOR correspond to the majority and XOR functions respectively.

In the above we used the loop for \( i \) in range\( (n) \) but we can expand this out by simply repeating the code \( n \) times, replacing the value of \( i \) with 0, 1, 2, ..., \( n - 1 \). The crucial point is that (unlike most programming languages) we do not allow the number of times the loop is executed to depend on the input, and so it is always possible to “expand out” the loop by simply copying the code the requisite number of times.

By expanding out all the features, for every value of \( n \) we can translate the above program into a standard (“sugar free”) NAND-CIRC program. Fig. 4.3 depicts what we get for \( n = 2 \).
By going through the above program carefully and accounting for the number of gates, we can see that it yields a proof of the following theorem (see also Fig. 4.4):

**Theorem 4.6 — Addition using NAND-CIRC programs.** For every $n \in \mathbb{N}$, let $ADD_n : \{0, 1\}^{2n} \rightarrow \{0, 1\}^{n+1}$ be the function that, given $x, x' \in \{0, 1\}^n$ computes the representation of the sum of the numbers that $x$ and $x'$ represent. Then there is a constant $c \leq 30$ such that for every $n$ there is a NAND-CIRC program of at most $c$ lines computing $ADD_n$.  

Once we have addition, we can use the grade-school algorithm to obtain multiplication as well, thus obtaining the following theorem:

**Theorem 4.7 — Multiplication using NAND-CIRC programs.** For every $n$, let $MULT_n : \{0, 1\}^{2n} \rightarrow \{0, 1\}^{2n}$ be the function that, given $x, x' \in \{0, 1\}^n$ computes the representation of the product of the numbers that $x$ and $x'$ represent. Then there is a constant $c$ such that for every $n$, there is a NAND-CIRC program of at most $cn^2$ lines computing $MULT_n$.

We omit the proof, though in Exercise 4.7 we ask you to supply a “constructive proof” in the form of a program (in your favorite programming language) that on input a number $n$, outputs the code of a NAND-CIRC program of at most $1000n^2$ lines that computes the $MULT_n$ function. In fact, we can use Karatsuba’s algorithm to show that there is a NAND-CIRC program of $O(n \log_2 3)$ lines to compute $MULT_n$ (and one can even get further asymptotic improvements using the newer algorithms).
4.3 THE LOOKUP FUNCTION

We have seen that NAND-CIRC programs can add and multiply numbers. But can they compute other type of functions, that have nothing to do with arithmetic? Here is one example:

**Definition 4.8 — Lookup function.** For every $k$, the lookup function $\text{LOOKUP}_k : \{0, 1\}^{2^k+k} \rightarrow \{0, 1\}$ is defined as follows: For every $x \in \{0, 1\}^{2^k}$ and $i \in \{0, 1\}^k$,

$$\text{LOOKUP}_k(x, i) = x_i \quad (4.1)$$

where $x_i$ denotes the $i^{th}$ entry of $x$, using the binary representation to identify $i$ with a number in $\{0, \ldots, 2^k - 1\}$.

See Fig. 4.5 for an illustration of the LOOKUP function. In particular the function $\text{LOOKUP}_1 : \{0, 1\}^3 \rightarrow \{0, 1\}$ maps $(x_0, x_1, i) \in \{0, 1\}^3$ to $x_i$. It is actually the same as the IF/MUX function we have seen above, that has a 4 line NAND-CIRC program. It turns out that for every $k$, we can compute $\text{LOOKUP}_k$ using a NAND-CIRC program:

**Theorem 4.9 — Lookup function.** For every $k > 0$, there is a NAND-CIRC program that computes the function $\text{LOOKUP}_k : \{0, 1\}^{2^k+k} \rightarrow \{0, 1\}$. Moreover, the number of lines in this program is at most $4 \cdot 2^k$.

An immediate corollary of Theorem 4.9 is that for every $k > 0$, $\text{LOOKUP}_k$ can be computed by a Boolean circuit (with AND, OR and NOT gates) of at most $8 \cdot 2^k$ gates.

4.3.1 Constructing a NAND-CIRC program for LOOKUP

We now prove Theorem 4.9. The idea is actually quite simple. Consider the function $\text{LOOKUP}_3 : \{0, 1\}^{2^3+3} \rightarrow \{0, 1\}$ that takes an input of $8 + 3 = 11$ bits and output a single bit. We can write this function in pseudocode as follows:
def LOOKUP3(X[0], X[1], X[2], X[3], X[4], X[5], X[6], X[7],
       i[0], i[1], i[2]):

    if i == (0, 0, 0): return X[0]
    if i == (0, 0, 1): return X[1]
    if i == (0, 1, 0): return X[2]

    ...
    if i == (1, 1, 1): return X[7]

A condition such as $i \Rightarrow (0, 1, 0)$ can be expanded out to the AND of 
NOT$(i[0])$, $i[1]$ and NOT$(i[2])$ and each one of these AND and NOT 
gates can be then translated into NAND. The above can yield a proof of 
a version of Theorem 4.9 with a slightly larger number of gates, but if 
we are a little more careful we can prove the theorem with the number 
of gates as stated.

Specifically, we will prove Theorem 4.9 by induction. We will do so 
by induction. That is, we show how to use a NAND-CIRC program 
for computing $LOOKUP_k$ to compute $LOOKUP_{k+1}$. For the case $k = 
1$, $LOOKUP_1$ is the same as IF for which we given a NAND-CIRC 
program with four line.

Now let us consider the case of $k = 2$. Given input 
\( x = (x_0, x_1, x_2, x_3) \) for $LOOKUP_2$ and an index $i = (i_0, i_1)$, if 
the most significant bit $i_0$ of the index is 0 then $LOOKUP_2(x, i)$ 
will equal $x_0$ if $i_1 = 0$ and equal $x_1$ if $i_1 = 1$. Similarly, if the most 
significant bit $i_0$ is 1 then $LOOKUP_2(x, i)$ will equal $x_2$ if $i_1 = 0$ and 
will equal $x_3$ if $i_1 = 1$. Another way to say this is that we can write 
$LOOKUP_2$ as follows:

def LOOKUP2(X[0], X[1], X[2], X[3], X[4], X[5], X[6], X[7],
       i[0], i[1], i[2]):

    if i[0] == 1:
        return LOOKUP1(X[2], X[3], i[1])
    else:
        return LOOKUP1(X[0], X[1], i[1])

or in other words,

def LOOKUP2(X[0], X[1], X[2], X[3], X[4], X[5], X[6], X[7],
       i[0], i[1], i[2]):

    a = LOOKUP1(X[2], X[3], i[1])
    b = LOOKUP1(X[0], X[1], i[1])
    return IF( i[0], a, b)

Similarly, we can write

def LOOKUP3(X[0], X[1], X[2], X[3], X[4], X[5], X[6], X[7],
       i[0], i[1], i[2]):
and so on and so forth. Generally, we can compute $LOOKUP_k$ using two invocations of $LOOKUP_{k-1}$ and one invocation of $IF$, which yields the following lemma:

**Lemma 4.10 — Lookup recursion.** For every $k \geq 2$, $LOOKUP_k(x_0, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-1})$ is equal to

$$IF(i_0, LOOKUP_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_1, \ldots, i_{k-1}), LOOKUP_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_1, \ldots, i_{k-1}))$$

(4.2)

**Proof.** If the most significant bit $i_0$ of $i$ is zero, then the index $i$ is in $\{0, \ldots, 2^{k-1} - 1\}$ and hence we can perform the lookup on the “first half” of $x$ and the result of $LOOKUP_k(x, i)$ will be the same as $a = LOOKUP_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_1, \ldots, i_{k-1})$. On the other hand, if this most significant bit $i_0$ is equal to 1, then the index is in $\{2^{k-1}, \ldots, 2^k - 1\}$, in which case the result of $LOOKUP_k(x, i)$ is the same as $b = LOOKUP_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_1, \ldots, i_{k-1})$. Thus we can compute $LOOKUP_k(x, i)$ by first computing $a$ and $b$ and then outputting

$$IF(i_0, a, b)$$

Lemma 4.10 directly implies Theorem 4.9. We prove by induction on $k$ that there is a NAND-CIRC program of at most $4 \cdot 2^k$ lines for $LOOKUP_k$. For $k = 1$ this follows by the four line program for $IF$ we’ve seen before. For $k > 1$, we use the following pseudocode

```plaintext
a = LOOKUP_{k-1}(X[0], \ldots, X[2^{k-1}-1], i[1], \ldots, i[k-1])
b = LOOKUP_{k-1}(X[2^k], \ldots, X[2^k+1], i[1], \ldots, i[k-1])
return IF(i[k-1], a, b)
```

If we let $L(k)$ be the number of lines required for $LOOKUP_k$, then the above shows that

$$L(k) \leq 2L(k-1) + 4$$

(4.3)

which solves for $L(k) \leq 4(2^k - 1)$. (See Fig. 4.6 for a plot of the actual number of lines in our implementation of $LOOKUP_k$.)

### 4.4 Computing Every Function

At this point we know the following facts about NAND-CIRC programs:

1. They can compute at least some non trivial functions.
2. Coming up with NAND-CIRC programs for various functions is a very tedious task.

Thus I would not blame the reader if they were not particularly looking forward to a long sequence of examples of functions that can be computed by NAND-CIRC programs. However, it turns out we are not going to need this, as we can show in one fell swoop that NAND-CIRC programs can compute every finite function:

**Theorem 4.11 — Universality of NAND.** There exists some constant $c > 0$ such that for every $n, m > 0$ and function $f : \{0, 1\}^n \to \{0, 1\}^m$, there is a NAND-CIRC program with at most $c \cdot m^2n$ lines that computes the function $f$.

By Theorem 3.20, the models of NAND circuits, NAND-CIRC programs, AON-CIRC programs, and Boolean circuits, are all equivalent to one another, and hence ?? holds for all these models. In particular, the following theorem is equivalent to ??:

**Theorem 4.12 — Universality of Boolean circuits.** There exists some constant $c > 0$ such that for every $n, m > 0$ and function $f : \{0, 1\}^n \to \{0, 1\}^m$, there is a Boolean circuit with at most $c \cdot m^2n$ gates that computes the function $f$.

**Big Idea 5** Every finite function can be computed by a large enough Boolean circuit.

**Remark 4.13 — Improved bound.** As we’ll see in the proof, the constant $c$ will be smaller than 10. In fact, with a tighter proof, we can even shave an extra factor of $n$, as well as optimize the constant, to obtain the following stronger result:

**Lemma 4.14** For every $\epsilon > 0$, $m \in \mathbb{N}$ and sufficiently large $n$, if $f : \{0, 1\}^n \to \{0, 1\}^m$ then $f$ can be computed by a NAND circuit of at most

$$\left(1 + \epsilon\right)\frac{m \cdot 2^n}{n} \quad (4.4)$$

gates.

We will not prove Lemma 4.14 in this book, but discuss how to obtain a bound of the form $O\left(\frac{m \cdot 2^n}{n}\right)$ in Section 4.4.2. See also the biographical notes.
4.4.1 Proof of NAND’s Universality

To prove Theorem 4.11, we need to give a NAND circuit, or equivalently a NAND-CIRC program, for every possible function. We will restrict our attention to the case of Boolean functions (i.e., $m = 1$). In Exercise 4.9 you will show how to extend the proof for all values of $m$.

A function $F : \{0, 1\}^n \to \{0, 1\}$ can be specified by a table of its values for each one of the $2^n$ inputs. For example, the table below describes one particular function $G : \{0, 1\}^4 \to \{0, 1\}$:

<table>
<thead>
<tr>
<th>Input ($x$)</th>
<th>Output ($G(x)$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
</tr>
<tr>
<td>0101</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>1001</td>
<td>0</td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>0</td>
</tr>
<tr>
<td>1100</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
</tr>
<tr>
<td>1110</td>
<td>1</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
</tr>
</tbody>
</table>

We can see that for every $x \in \{0, 1\}^4$, $G(x) = \text{LOOKUP}_4(1100100100001111, x)$. Therefore the following is NAND “pseudocode” to compute $G$:

```plaintext
G0000 = 1
G1000 = 1
G0100 = 0
...
G0111 = 1
G1111 = 1
Y[0] = \text{LOOKUP}_4(G0000, G1000, \ldots, G1111, 
X[0], X[1], X[2], X[3])
```

We can translate this pseudocode into an actual NAND-CIRC program by adding three lines to define variables zero and one that are

\[^{7}\text{In case you are curious, this is the function on input}
\]
\[^{7}\text{$i \in \{0, 1\}^4$ (which we interpret as a number in [16],}
\]
\[^{7}\text{outputs the $i$-th digit of } \pi \text{ in the binary basis.}\]
initialized to 0 and 1 respectively, and then replacing a statement such as $G_{xx} = 0$ with $G_{xx} = \text{NAND}(\text{one}, \text{one})$ and a statement such as $G_{xx} = 1$ with $G_{xx} = \text{NAND}(\text{zero}, \text{zero})$. The call to LOOKUP_4 will be replaced by the NAND-CIRC program that computes $\text{LOOKUP}_4$, plugging in the appropriate inputs.

There was nothing about the above reasoning that was particular to the function $G$ of $??$. Given every function $F : \{0, 1\}^n \rightarrow \{0, 1\}$, we can write a NAND-CIRC program that does the following:

1. Initialize $2^n$ variables of the form $F00\ldots 0$ till $F11\ldots 1$ so that for every $z \in \{0, 1\}^n$, the variable corresponding to $z$ is assigned the value $F(z)$.

2. Compute $\text{LOOKUP}_n$ on the $2^n$ variables initialized in the previous step, with the index variable being the input variables $X[0], \ldots, X[2^n - 1]$. That is, just like in the pseudocode for $G$ above, we use $Y[0] = \text{LOOKUP}(F00\ldots 0, \ldots, F11\ldots 1, X[0], \ldots, X[n − 1])$

The total number of lines in the program will be $2^n + 4 \cdot 2^n$ lines that we pay for computing $\text{LOOKUP}_n$. This completes the proof of Theorem 4.11.

---

**Remark 4.15 — Result in perspective.** While Theorem 4.11 seems striking at first, in retrospect, it is perhaps not that surprising that every finite function can be computed with a NAND-CIRC program. After all, a finite function $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$ can be represented by simply the list of its outputs for each one of the $2^n$ input values. So it makes sense that we could write a NAND-CIRC program of similar size to compute it. What is more interesting is that some functions, such as addition and multiplication, have a much more efficient representation: one that only requires $O(n^2)$ or even smaller number of lines.

### 4.4.2 Improving by a factor of $n$ (optional)

As discussed in Remark 4.13, by being a little more careful, we can improve the bound of Theorem 4.11 and show that every function $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$ can be computed by a NAND-CIRC program of at most $O(m 2^n / n)$ lines. As before, it is enough to prove the case that $m = 1$.

The idea is to use the technique known as memoization. Let $k = \log(n - 2 \log n)$ (the reasoning behind this choice will become clear later on). For every $a \in \{0, 1\}^{n−k}$ we define $F_a : \{0, 1\}^k \rightarrow \{0, 1\}$ to be the function that maps $w_0, \ldots, w_{k−1}$ to $F(a_0, \ldots, a_{n−k−1}, w_0, \ldots, w_{k−1})$. 
On input \( x = x_0, \ldots, x_{n-1} \), we can compute \( F(x) \) as follows: First we compute a \( 2^n \cdot 2^k \) long string \( P \) whose \( a^{th} \) entry (identifying \( \{0, 1\}^n \) with \( \{2^n\}^k \)) equals \( F_a(x_{n-k}, \ldots, x_{n-1}) \). One can verify that \( F(x) = \text{LOOKUP}_{n-k}(P, x_0, \ldots, x_{n-k-1}) \). Since we can compute \( \text{LOOKUP}_{n-k} \) using \( O(2^n \cdot 2^k) \) lines, if we can compute the string \( P \) (i.e., compute variables \( P_0, \ldots, P_{2^n-k-1} \)) using \( T \) lines, then we can compute \( F \) in \( O(2^n \cdot T) \) lines.

The trivial way to compute the string \( P \) would be to use \( O(2^n) \) lines to compute for every \( a \) the map \( x_0, \ldots, x_{k-1} \mapsto F_a(x_0, \ldots, x_{k-1}) \) as in the proof of Theorem 4.11. Since there are \( 2^n-k \) \( a \)'s, that would be a total cost of \( O(2^n \cdot 2^k) = O(2^n) \) which would not improve at all on the bound of Theorem 4.11. However, a more careful observation shows that we are making some redundant computations. After all, there are only \( 2^k \) distinct functions mapping \( k \) bits to one bit. If \( a \) and \( a' \) satisfy that \( F_a = F_{a'} \), then we don’t need to spend \( 2^k \) lines computing both \( F_a(x) \) and \( F_{a'}(x) \) but rather can only compute the variable \( P_{a} \) and then copy \( P_{a} \) to \( P_{a'} \) using \( O(1) \) lines. Since we have \( 2^k \) unique functions, we can bound the total cost to compute \( P \) by \( O(2^k \cdot 2^k) + O(2^n-k) \).

Now it just becomes a matter of calculation. By our choice of \( k \), \( 2^k = n - 2 \log n \) and hence \( 2^k = \frac{2^n}{n} \). Since \( n/2 \leq 2^k \leq n \), we can bound the total cost of computing \( F(x) \) (including also the additional \( O(2^n-k) \) cost of computing \( \text{LOOKUP}_{n-k} \)) by \( O(2^n/n) + O(2^n/n) \), which is what we wanted to prove.

### 4.5 Computing Every Function: An Alternative Proof

Theorem 4.12 is a fundamental result in the theory (and practice!) of computation. In this section we present an alternative proof of this basic fact that Boolean circuits can compute every finite function. This alternative proof gives somewhat worse quantitative bound on the number of gates but it has the advantage of being simpler, working directly with circuits and avoiding the usage of all the syntactic sugar machinery. (However, that machinery is useful in its own right, and will find other applications later on.)

**Theorem 4.16 — Universality of Boolean circuits (alternative phrasing).** There exists some constant \( c > 0 \) such that for every \( n, m > 0 \) and function \( f : \{0, 1\}^n \rightarrow \{0, 1\}^m \), there is a Boolean circuit with at most \( c \cdot m \cdot n 2^n \) gates that computes the function \( f \).

**Proof Idea:**

The idea of the proof is illustrated in Fig. 4.7. As before, it is enough to focus on the case that \( m = 1 \) (the function \( f \) has a single output), since we can always extend this to the case of \( m > 1 \) by looking at...
the composition of \( m \) circuits each computing a different output bit of the function \( f \). We start by showing that for every \( \alpha \in \{0, 1\}^n \), there is an \( O(n) \) sized circuit that computes the function \( \delta_{\alpha} : \{0, 1\}^n \rightarrow \{0, 1\} \) defined as follows: \( \delta_{\alpha}(x) = 1 \) iff \( x = \alpha \) (that is, \( \delta_{\alpha} \) outputs 0 on all inputs except the input \( \alpha \)). We can then write any function \( f : \{0, 1\}^n \rightarrow \{0, 1\} \) as the OR of at most \( 2^n \) functions \( \delta_{\alpha} \) for the \( \alpha \)'s on which \( f(\alpha) = 1 \).

\[ f(\alpha) = \delta_{x_0}(\alpha) \vee \delta_{x_1}(\alpha) \vee \cdots \vee \delta_{x_{n-1}}(\alpha) \tag{4.6} \]

where \( S = \{x_0, \ldots, x_{n-1}\} \) is the set of inputs on which \( f \) outputs 1. (Indeed one can verify that the right-hand side of (4.6) evaluates to 1 on \( x \in \{0, 1\}^n \) if and only if \( x \) is in the set \( S \).)

Proof of Theorem 4.16. As noted above, we prove the theorem for the case \( m = 1 \). The result can be extended for \( m > 1 \) as before (see also Exercise 4.9). Let \( f : \{0, 1\}^n \rightarrow \{0, 1\} \). We will prove that there is an \( O(n \cdot 2^n) \)-sized Boolean circuit to compute \( f \) in the following steps:

1. We start by showing that for every \( \alpha \in \{0, 1\}^n \), there is an \( O(n) \) sized circuit that computes the function \( \delta_{\alpha} : \{0, 1\}^n \rightarrow \{0, 1\} \), where \( \delta_{\alpha}(x) = 1 \) iff \( x = \alpha \).

2. We then show that this implies the existence of an \( O(n \cdot 2^n) \)-sized circuit that computes \( f \), by writing \( f(x) \) as the OR of \( \delta_{\alpha}(x) \) for all \( \alpha \in \{0, 1\}^n \) such that \( f(\alpha) = 1 \).

We start in Step 1:

**CLAIM:** For \( \alpha \in \{0, 1\}^n \), define \( \delta_{\alpha} : \{0, 1\}^n \) as follows:

\[
\delta_{\alpha}(x) = \begin{cases} 
1 & x = \alpha \\
0 & \text{otherwise}
\end{cases} \tag{4.5}
\]

then there is a Boolean circuit using at most \( 2n \) gates that computes \( \delta_{\alpha} \).

**PROOF OF CLAIM:** The proof is illustrated in Fig. 4.8. As an example, consider the function \( \delta_{011} : \{0, 1\}^3 \rightarrow \{0, 1\} \). This function outputs 1 on \( x \) if and only if \( x_0 = 0, x_1 = 1 \) and \( x_2 = 1 \), and so we can write \( \delta_{011}(x) = \overline{x_0} \wedge x_1 \wedge x_2 \), which translates into a Boolean circuit with one NOT gate and two AND gates. More generally, for every \( \alpha \in \{0, 1\}^n \), we can express \( \delta_{\alpha}(x) \) as \( (x_0 = \alpha_0) \wedge (x_1 = \alpha_1) \wedge \cdots \wedge (x_{n-1} = \alpha_{n-1}) \), where if \( \alpha_i = 0 \) we replace \( x_i = \alpha_i \) with \( \overline{x_i} \), and if \( \alpha_i = 1 \) we replace \( x_i = \alpha_i \) by simply \( x_i \). This yields a circuit that computes \( \delta_{\alpha} \) using \( n \) AND gates and at most \( n \) NOT gates and so a total of at most \( 2n \) gates.

Now for every function \( f : \{0, 1\}^n \rightarrow \{0, 1\} \), we can write

\[
f(x) = \delta_{x_0}(x) \vee \delta_{x_1}(x) \vee \cdots \vee \delta_{x_{n-1}}(x)
\]

where \( S = \{x_0, \ldots, x_{n-1}\} \) is the set of inputs on which \( f \) outputs 1. (Indeed one can verify that the right-hand side of (4.6) evaluates to 1 on \( x \in \{0, 1\}^n \) if and only if \( x \) is in the set \( S \).)
Therefore we can compute $f$ using a Boolean circuit of at most $2n$ gates for each of the $N$ functions $\delta_\alpha$, and combine that with at most $N$ OR gates, thus obtaining a circuit of at most $2n \cdot N + N$ gates. Since $S \subseteq \{0, 1\}^n$, its size $N$ is at most $2^n$ and hence the total number of gates in this circuit is $O(n \cdot 2^n)$.

4.6 THE CLASS SIZE$_{n,m}(T)$

We have seen that every function $f : \{0, 1\}^n \to \{0, 1\}^m$ can be computed by a circuit of size $O(m \cdot 2^n)$, and some functions (such as addition and multiplication) can be computed by much smaller circuits. This motivates the following definition which we will find very useful later on in this book. We define SIZE$_{n,m}(s)$ to be the set of all functions from $\{0, 1\}^n$ to $\{0, 1\}^m$ that can be computed by NAND circuits of at most $s$ gates (or equivalently, by NAND-CIRC programs of at most $s$ lines). In other words, SIZE$_{n,m}(s)$ is defined as follows:

**Definition 4.17 — Size class of functions.** Let $n, m, s \in \mathbb{N}$ be numbers with $s \geq m$. The set SIZE$_{n,m}(s)$ denotes the set of all functions $f : \{0, 1\}^n \to \{0, 1\}^m$ such that there exists a NAND circuit of at most $s$ gates that computes $f$. We denote by SIZE$_n(s)$ the set SIZE$_{n,1}(s)$.

Fig. 4.9 depicts the sets SIZE$_{n,1}(s)$, note that SIZE$_{n,m}(s)$ is a set of functions, not of programs! (asking if a program or a circuit is a member of SIZE$_{n,m}(s)$ is a category error as in the sense of Fig. 4.10).

**Theorem 4.11** shows that every function $g$ is computable by some circuit of at most $c \cdot 2^n/n$ gates, and hence SIZE$_{n,1}(c \cdot 2^n/n)$ corresponds to the set of all functions from $\{0, 1\}^n$ to $\{0, 1\}$.
While we defined $\text{SIZE}_{n,m}(s)$ with respect to NAND gates, we would get essentially the same class if we defined it with respect to AND/OR/NOT gates:

**Lemma 4.18** Let $\text{SIZE}^{\text{AON}}_{n,m,s}$ denote the set of all functions $f : \{0,1\}^n \to \{0,1\}^m$ that can be computed by an AND/OR/NOT Boolean circuit of at most $s$ gates. Then,

$$\text{SIZE}_{n,m}(s/2) \subseteq \text{SIZE}^{\text{AON}}_{n,m}(s) \subseteq \text{SIZE}_{n,m}(3s) \quad (4.7)$$

**Proof.** If $f$ can be computed by a NAND circuit of at most $s/2$ gates, then by replacing each NAND with the two gates NOT and AND, we can obtain an AND/OR/NOT Boolean circuit of at most $s$ gates that computes $f$. On the other hand, if $f$ can be computed by a Boolean AND/OR/NOT circuit of at most $s$ gates, then by Theorem 3.13 it can be computed by a NAND circuit of at most $3s$ gates.

The results we’ve seen before can be phrased as showing that $\text{ADD}_n \in \text{SIZE}_{2n,n+1}(100n)$ and $\text{MULT}_n \in \text{SIZE}_{2n,2n}(10000n \log_2 3)$. Theorem 4.11 shows that $\text{SIZE}_{n,m}(4m2^n)$ is equal the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$. See Fig. 5.4.

---

**Remark 4.19 — Finite vs infinite functions.** A NAND-CIRC program $P$ can only compute a function with a certain number $n$ of inputs and a certain number $m$ of outputs. Hence for example there is no single NAND-CIRC program that can compute the increment function $\text{INC} : \{0,1\}^* \to \{0,1\}^*$ that maps a string $x$ (which we identify with a number via the binary representation) to the string that represents $x + 1$. Rather for every $n > 0$, there is a NAND-CIRC program $P_n$ that computes the restriction $\text{INC}_n$ of the function $\text{INC}$ to inputs of length $n$. Since it can be shown that for every $n > 0$ such a program $P_n$ exists of length at most $10n$, $\text{INC}_n \in \text{SIZE}(10n)$ for every $n > 0$.

If $T : \mathbb{N} \to \mathbb{N}$ and $F : \{0,1\}^* \to \{0,1\}^*$, we will sometimes slightly abuse notation and write $\in \text{EVEN}$?
F ∈ SIZE(T(n)) to indicate that for every n the restriction F↾n of F to inputs in \{0, 1\}^n is in SIZE(T(n)). Hence we can write INC ∈ SIZE(10n). We will come back to this issue of finite vs infinite functions later in this course.

Solved Exercise 4.1 — SIZE closed under complement. In this exercise we prove a certain "closure property" of the class SIZE_n(s). That is, we show that if f is in this class then (up to some small additive term) so is the complement of f, which is the function g(x) = 1 − f(x).

Prove that there is a constant c such that for every f : \{0, 1\}^n → \{0, 1\} and s ∈ \mathbb{N}, if f ∈ SIZE_n(s) then 1 − f ∈ SIZE_n(s + c).

Solution:
If f ∈ SIZE(s) then there is an s-line NAND-CIRC program P that computes f. We can rename the variable Y[0] in P to a variable temp and add the line

Y[0] = NAND(temp,temp)

at the very end to obtain a program P' that computes 1 − f.

Lecture Recap
- We can define the notion of computing a function via a simplified "programming language", where computing a function F in T steps would correspond to having a T-line NAND-CIRC program that computes F.
- While the NAND-CIRC programming only has one operation, other operations such as functions and conditional execution can be implemented using it.
- Every function f : \{0, 1\}^n → \{0, 1\}^m can be computed by a circuit of at most O(m2^n) gates (and in fact at most O(m2^n/n) gates).
- Sometimes (or maybe always?) we can translate an efficient algorithm to compute f into a circuit that computes f with a number of gates comparable to the number of steps in this algorithm.

4.7 EXERCISES
Exercise 4.1 — Pairing. This exercise asks you to give a one-to-one map from \mathbb{N}^2 to \mathbb{N}. This can be useful to implement two-dimensional arrays...
as “syntactic sugar” in programming languages that only have one-dimensional array.

1. Prove that the map $F(x, y) = 2^x 3^y$ is a one-to-one map from $\mathbb{N}^2$ to $\mathbb{N}$.

2. Show that there is a one-to-one map $F : \mathbb{N}^2 \to \mathbb{N}$ such that for every $x, y$, $F(x, y) \leq 100 \cdot \max\{x, y\}^2 + 100$.

3. For every $k$, show that there is a one-to-one map $F : \mathbb{N}^k \to \mathbb{N}$ such that for every $x_0, \ldots, x_{k-1} \in \mathbb{N}$, $F(x_0, \ldots, x_{k-1}) \leq 100 \cdot (x_0 + x_1 + \ldots + x_{k-1} + 100k)^k$.

Exercise 4.2 — Computing MUX. Prove that the NAND-CIRC program below computes the function MUX (or LOOKUP) where MUX($a, b, c$) equals $a$ if $c = 0$ and equals $b$ if $c = 1$:

\[
\begin{align*}
t &= \text{NAND}(X[2], X[2]) \\
u &= \text{NAND}(X[0], t) \\
v &= \text{NAND}(X[1], X[2]) \\
Y[0] &= \text{NAND}(u, v)
\end{align*}
\]

Exercise 4.3 — At least two / Majority. Give a NAND-CIRC program of at most 6 lines to compute MAJ : $\{0, 1\}^3 \to \{0, 1\}$ where MAJ($a, b, c$) equals 1 iff $a + b + c \geq 2$.

Exercise 4.4 — Conditional statements. In this exercise we will show that even though the NAND-CIRC programming language does not have an if .. then .. else .. statement, we can still implement it. Suppose that there is an $s$-line NAND-CIRC program to compute $f : \{0, 1\}^n \to \{0, 1\}$ and an $s'$-line NAND-CIRC program to compute $f' : \{0, 1\}^n \to \{0, 1\}$. Prove that there is a program of at most $s + s' + 10 + 10$ lines to compute the function $g : \{0, 1\}^{n+1} \to \{0, 1\}$ where $g(x_0, \ldots, x_{n-1}, x_n)$ equals $f(x_0, \ldots, x_{n-1})$ if $x_n = 0$ and equals $f'(x_0, \ldots, x_{n-1})$ otherwise.

Exercise 4.5 — Half and full adders. 1. A half adder is the function $HA : \{0, 1\}^2 \to \{0, 1\}^2$ that corresponds to adding two binary bits. That is, for every $a, b \in \{0, 1\}$, $HA(a, b) = (e, f)$ where $2e + f = a + b$.

Prove that there is a NAND circuit of at most five NAND gates that computes $HA$.

2. A full adder is the function $FA : \{0, 1\}^3 \to \{0, 1\}$ that takes in two bits and a “carry” bit and outputs their sum. That is, for every
a, b, c ∈ \{0, 1\}, \text{FA}(a, b, c) = (e, f) such that \(2e + f = a + b + c\).

Prove that there is a NAND circuit of at most nine NAND gates that computes \(\text{FA}\).

3. Prove that if there is a NAND circuit of \(c\) gates that computes \(\text{FA}\), then there is a circuit of \(cn\) gates that computes \(\text{ADD}_n\) where (as in Theorem 4.6) \(\text{ADD}_n : \{0, 1\}^{2n} \rightarrow \{0, 1\}^n\) is the function that outputs the addition of two input \(n\)-bit numbers. See footnote for hint.\(^9\)

4. Show that for every \(n\) there is a NAND-CIRC program to compute \(\text{ADD}_n\) with at most \(9n\) lines.

**Exercise 4.6 — Addition.** Write a program using your favorite programming language that on input an integer \(n\), outputs a NAND-CIRC program that computes \(\text{ADD}_n\). Can you ensure that the program it outputs for \(\text{ADD}_n\) has fewer than \(10n\) lines?

**Exercise 4.7 — Multiplication.** Write a program using your favorite programming language that on input an integer \(n\), outputs a NAND-CIRC program that computes \(\text{MULT}_n\). Can you ensure that the program it outputs for \(\text{MULT}_n\) has fewer than \(1000 \cdot n^2\) lines?

**Exercise 4.8 — Efficient multiplication (challenge).** Write a program using your favorite programming language that on input an integer \(n\), outputs a NAND-CIRC program that computes \(\text{MULT}_n\) and has at most \(10000n^{1.9}\) lines.\(^{10}\) What is the smallest number of lines you can use to multiply two 2048 bit numbers?

**Exercise 4.9 — Multibit function.** Prove that

\(a.\) If there is an \(s\)-line NAND-CIRC program to compute \(f : \{0, 1\}^n \rightarrow \{0, 1\}\) and an \(s'\)-line NAND-CIRC program to compute \(f' : \{0, 1\}^n \rightarrow \{0, 1\}\) then there is an \(s + s'\)-line program to compute the function \(g : \{0, 1\}^n \rightarrow \{0, 1\}^2\) such that \(g(x) = (f(x), f'(x))\).

\(b.\) For every function \(f : \{0, 1\}^n \rightarrow \{0, 1\}^m\), there is a NAND-CIRC program of at most \(10m \cdot 2^n\) lines that computes \(f\).

**Exercise 4.10 — Simplifying using syntactic sugar.** Let \(P\) be the following NAND-CIRC program:

\[
\text{Temp}[0] = \text{NAND}(X[0], X[0]) \\
\text{Temp}[1] = \text{NAND}(X[1], X[1])
\]
Temp[2] = NAND(Temp[0],Temp[1])
Temp[3] = NAND(X[2],X[2])
Temp[4] = NAND(X[3],X[3])
Temp[5] = NAND(Temp[3],Temp[4])
Temp[6] = NAND(Temp[2],Temp[2])
Temp[7] = NAND(Temp[5],Temp[5])
Y[0] = NAND(Temp[6],Temp[7])

1. Write a program $P'$ with at most three lines of code that uses both NAND as well as the syntactic sugar OR that computes the same function as $P$.

2. Draw a circuit that computes the same function as $P$ and uses only AND and NOT gates.

In the following exercises you are asked to compare the power of pairs programming languages. By “comparing the power” of two programming languages $X$ and $Y$ we mean determining the relation between the set of functions that are computable using programs in $X$ and $Y$ respectively. That is, to answer such a question you need to do both of:

1. Either prove that for every program $P$ in $X$ there is a program $P'$ in $Y$ that computes the same function as $P$, or give an example for a function that is computable by an $X$-program but not computable by a $Y$-program.

and

1. Either prove that for every program $P$ in $Y$ there is a program $P'$ in $X$ that computes the same function as $P$, or give an example for a function that is computable by a $Y$-program but not computable by an $X$-program.

When you give an example as above of a function that is computable in one programming language but not the other, you need to prove that the function you showed is (1) computable in the first programming language, (2) not computable in the second programming language.

**Exercise 4.11 — Compare IF and NAND.** Let IF-CIRC be the programming language where we have the following operations foo = 0, foo = 1, foo = IF(cond, yes, no) (that is, we can use the constants 0 and 1, and the IF : $\{0, 1\}^3 \to \{0, 1\}$ function such that $IF(a, b, c)$ equals $b$ if $a = 1$ and equals $c$ if $a = 0$). Compare the power of the NAND-CIRC programming language and the IF-CIRC programming language.
Exercise 4.12 — Compare XOR and NAND. Let XOR-CIRC be the programming language where we have the following operations f oo = XOR(bar, blah), f oo = 1 and bar = 0 (that is, we can use the constants 0, 1 and the XOR function that maps a, b ∈ {0, 1}^2 to a + b mod 2). Compare the power of the NAND-CIRC programming language and the XOR-CIRC programming language.\footnote{Hint: You can use the fact that (a + b) + c mod 2 = a + b + c mod 2. In particular it means that if you have the lines d = XOR(a, b) and e = XOR(d, c) then e gets the sum modulo 2 of the variable a, b and c.}

Exercise 4.13 — Circuits for majority. Prove that there is some constant c such that for every n > 1, MAJ_n ∈ Size(cn) where MAJ_n : {0, 1}^n → {0, 1} is the majority function on n input bits. That is MAJ_n(x) = 1 iff \(\sum_{i=0}^{n-1} x_i > n/2\). NOTE: You can get 16 points by proving the weaker statement MAJ_n ∈ Size(c · n log n) for some constant c.\footnote{Hint: One approach to solve this is using recursion and the so-called Master Theorem.}

Exercise 4.14 — Circuits for threshold. Prove that there is some constant c such that for every n > 1, and integers a_0, … , a_{n-1}, b ∈ \{-2^n, -2^n + 1, … , -1, 0, +1, … , 2^n\}, there is a NAND circuit with at most n^c gates that computes the threshold function f_{a_0, … , a_{n-1}, b} : {0, 1}^n → {0, 1} that on input x ∈ {0, 1}^n outputs 1 if and only if \(\sum_{i=0}^{n-1} a_i x_i > b\).

4.8 BIBLIOGRAPHICAL NOTES

See Jukna’s and Wegener’s books [Juk12; Weg87] for much more extensive discussion on circuits. Shannon showed that every Boolean function can be computed by a circuit of exponential size [Sha38]. The improved bound of \(c \cdot 2^n/n\) (with the optimal value of c for many bases) is due to Lupanov [Lup58]. An exposition of this for the case of NAND is given in Chapter 4 of his book [Lup84]. (Thanks to Sasha Golovnev for tracking down this reference!)