Learning Objectives:

- Get comfort with syntactic sugar or automatic translation of higher level logic to NAND code.
- More techniques for translating informal or higher level language algorithms into NAND.
- Learn proof of major result: every finite function can be computed by some NAND program.
- Start thinking *quantitatively* about number of lines required for computation.

4

Syntactic sugar, and computing every function

> *"[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 NAND programing language is pretty much as "bare bones" as programming languages come. After all, it only has a single operation. But, it turns out we can implement some "added features" on top of it. That is, we can show how we can implement those features using the underlying mechanisms of the language.

Let's start with a simple example. One of the most basic operations a programming language has is to assign the value of one variable into another. And yet in NAND, we cannot even do that, as we only allow assignments of the result of a NAND operation. Yet, it is possible to "pretend" that we have such an assignment operation, by transforming code such as

```
foo = COPY(bar)
```

into the valid NAND code:

```
notbar = NAND(bar,bar)
foo    = NAND(notbar,notbar)
```

the reason being that for every $a \in \{0,1\}$, $NAND(a,a) = NOT(aNANDa) = NOT(a)$ and so in these two lines `notbar` is assigned the negation of `bar` and so `foo` is assigned the negation of the negation of `bar`, which is simply `bar`.

Thus in describing NAND programs we can (and will) allow ourselves to use the variable assignment operation, with the understanding that in actual programs we will replace every line of the first form with the two lines of the second form. In programming language parlance this is known as "syntactic sugar", since we are not changing the definition of the language, but merely introducing some convenient notational shortcuts.[1] We will use several such "syntactic sugar" constructs to make our descriptions of NAND programs shorter and simpler. However, these descriptions are merely shorthand for the equivalent standard or "sugar free" NAND program that is obtained after removing the use of all these constructs. In particular, when we say that a function F has an s-line NAND program, we mean a standard NAND program, that does not use any syntactic sugar. The website http://www.nandpl.org contains an online "unsweetener" that can take a NAND program that uses these features and modifies it to an equivalent program that does not use them.

[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.

4.1 SOME USEFUL SYNTACTIC SUGAR

In this section, we will list some additional examples of "syntactic sugar" transformations. Going over all these examples can be somewhat tedious, but we do it for two reasons:

1. To convince you that despite its seeming simplicity and limitations, the NAND 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… `:)`

4.1.1 Constants

We can create variables `zero` and `one` that have the values 0 and 1 respectively by adding the lines

```
temp = NAND(X[0],X[0])
one  = NAND(temp,X[0])
zero = NAND(one,one)
```

Note that since for every $x \in \{0,1\}$, $NAND(x,\overline{x}) = 1$, the variable `one` will get the value 1 regardless of the value of x_0, and the variable `zero` will get the value $NAND(1,1) = 0$.[2] We can combine the above two techniques to enable assigning constants to variables in our programs.

[2] We could have saved a couple of lines using the convention that uninitialized variables default to 0, but it's always nice to be explicit.

4.1.2 Functions / Macros

Another staple of almost any programming language is the ability to 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 such as

```
def Func(a,b):
    function_code
    return c
some_code
f = Func(e,d)
some_more_code
```

```
some_code
function_code'
some_more_code
```

where `function_code'` is obtained by replacing all occurrences of `a` with `d`,`b` 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 by replacing every instance of a variable `foo` with `upfoo` where `up` is some unique prefix.

4.1.3 Example: Computing Majority via NAND's

Function definition allow us to express NAND programs much more cleanly and succinctly. For example, because we can compute AND,OR, NOT using NANDs, we can compute the *Majority* function as well.

```
def NOT(a): return NAND(a,a)
def AND(a,b): return NOT(NAND(a,b))
def OR(a,b): return NAND(NOT(a),NOT(b))

def MAJ(a,b,c):
    return OR(OR(AND(a,b),AND(b,c)),AND(a,c))

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

This is certainly much more pleasant than the full NAND alternative:

```
Temp[0] = NAND(X[0],X[1])
Temp[1] = NAND(Temp[0],Temp[0])
Temp[2] = NAND(X[1],X[2])
Temp[3] = NAND(Temp[2],Temp[2])
```

```
Temp[4] = NAND(Temp[1],Temp[1])
Temp[5] = NAND(Temp[3],Temp[3])
Temp[6] = NAND(Temp[4],Temp[5])
Temp[7] = NAND(X[0],X[2])
Temp[8] = NAND(Temp[7],Temp[7])
Temp[9] = NAND(Temp[6],Temp[6])
Temp[10] = NAND(Temp[8],Temp[8])
Y[0] = NAND(Temp[9],Temp[10])
```

4.1.4 Conditional statements

Another sorely missing feature in NAND 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 $IF : \{0,1\}^3 \rightarrow \{0,1\}$ such that $IF(a,b,c)$ equals b if $a = 1$ and c if $a = 0$.

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

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


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. We leave it as Exercise 4.2 to verify that this program does indeed compute this function.

Using the IF function, we can implement conditionals in NAND: To achieve something like

```
if (cond):
    a = ...
    b = ...
    c = ...
```

we can use code of the following form

```
a = IF(cond,...,a)
b = IF(cond,...,b)
c = IF(cond,...,c)
```

or even

```
a,b,c = IF(cond,.....,a,b,c)
```

using an extension of the IF function to more inputs and outputs.

4.1.5 Bounded loops

We can use "copy paste" to implement a bounded variant of *loops*, as long we only need to repeat the loop a fixed number of times. For example, we can use code such as:

```
for i in [7,9,12]:
    Foo[i] = NAND(Bar[2*i],Blah[3*i+1])
```

as shorthand for

```
Foo[7]  = NAND(Bar[14],Blah[22])
Foo[9]  = NAND(Bar[18],Blah[28])
Foo[12] = NAND(Bar[24],Blah[37])
```

One can also consider fancier versions, including inner loops and so on. 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. We will use standard Python syntax such as `range(n)` for the sets we can range over.

4.1.6 Example: Adding two integers

Using the above features, we can write the integer addition function as follows:

```
# Add two n-bit integers
def ADD(A,B):
    n = len(A)
    Result = [0]*(n+1)
    Carry  = [0]*(n+1)
    Carry[0] = zero(A[0])
    for i in range(n):
        Result[i] = XOR(Carry[i],XOR(A[i],B[i]))
        Carry[i+1] = MAJ(Carry[i],A[i],B[i])
    Result[n] = Carry[n]
    return Result
```

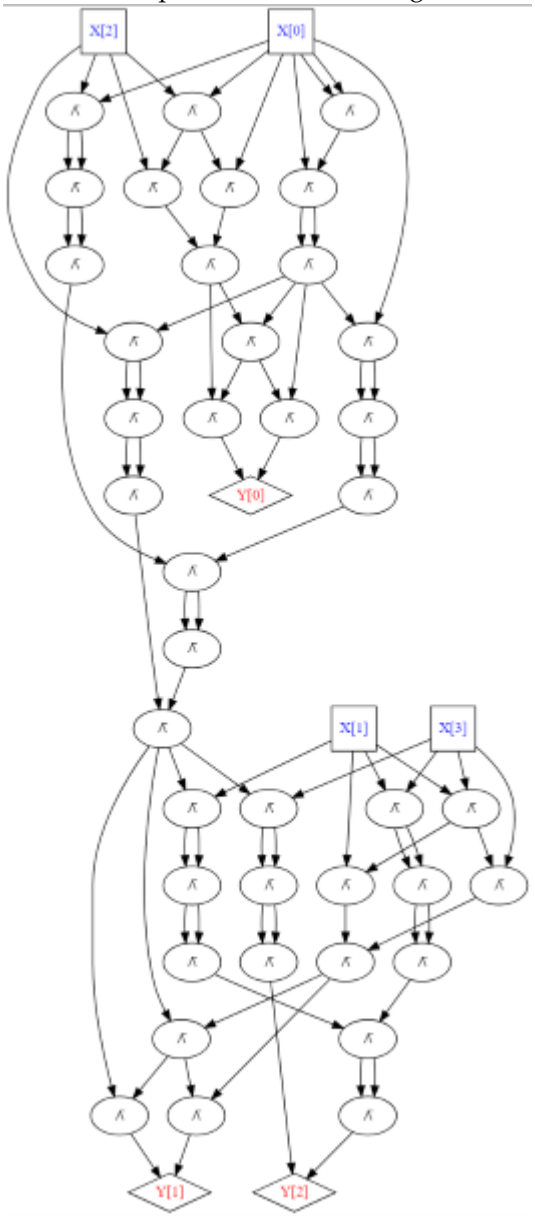
```
ADD([1,1,1,0,0],[1,0,0,0,0])
# [0, 0, 0, 1, 0, 0]
```

where `zero` is the constant zero function, and `MAJ` and `XOR` correspond to the majority and XOR functions respectively. This "sugared" version is certainly easier to read than even the two bit NAND addition program (obtained by restricting the above to the case $n = 2$):

```
Temp[0] = NAND(X[0],X[0])
Temp[1] = NAND(X[0],Temp[0])
Temp[2] = NAND(Temp[1],Temp[1])
Temp[3] = NAND(X[0],X[2])
Temp[4] = NAND(X[0],Temp[3])
Temp[5] = NAND(X[2],Temp[3])
Temp[6] = NAND(Temp[4],Temp[5])
Temp[7] = NAND(Temp[2],Temp[6])
Temp[8] = NAND(Temp[2],Temp[7])
Temp[9] = NAND(Temp[6],Temp[7])
Y[0] = NAND(Temp[8],Temp[9])
Temp[11] = NAND(Temp[2],X[0])
Temp[12] = NAND(Temp[11],Temp[11])
Temp[13] = NAND(X[0],X[2])
Temp[14] = NAND(Temp[13],Temp[13])
Temp[15] = NAND(Temp[12],Temp[12])
Temp[16] = NAND(Temp[14],Temp[14])
Temp[17] = NAND(Temp[15],Temp[16])
Temp[18] = NAND(Temp[2],X[2])
Temp[19] = NAND(Temp[18],Temp[18])
Temp[20] = NAND(Temp[17],Temp[17])
Temp[21] = NAND(Temp[19],Temp[19])
Temp[22] = NAND(Temp[20],Temp[21])
Temp[23] = NAND(X[1],X[3])
Temp[24] = NAND(X[1],Temp[23])
Temp[25] = NAND(X[3],Temp[23])
Temp[26] = NAND(Temp[24],Temp[25])
Temp[27] = NAND(Temp[22],Temp[26])
Temp[28] = NAND(Temp[22],Temp[27])
Temp[29] = NAND(Temp[26],Temp[27])
Y[1] = NAND(Temp[28],Temp[29])
Temp[31] = NAND(Temp[22],X[1])
Temp[32] = NAND(Temp[31],Temp[31])
Temp[33] = NAND(X[1],X[3])
Temp[34] = NAND(Temp[33],Temp[33])
Temp[35] = NAND(Temp[32],Temp[32])
```

```
Temp[36] = NAND(Temp[34],Temp[34])
Temp[37] = NAND(Temp[35],Temp[36])
Temp[38] = NAND(Temp[22],X[3])
Temp[39] = NAND(Temp[38],Temp[38])
Temp[40] = NAND(Temp[37],Temp[37])
Temp[41] = NAND(Temp[39],Temp[39])
Y[2] = NAND(Temp[40],Temp[41])
```

Which corresponds to the following circuit:



4.2 EVEN MORE SUGAR (OPTIONAL)

We can go even beyond this, and add more "syntactic sugar" to NAND. The key observation is that all of these are *not* extra features to NAND, but only ways that make it easier for us to write programs.

4.2.1 More indices

As stated, the NAND programming language only allows for "one dimensional arrays", in the sense that we can use variables such as `Foo[7]` or `Foo[29]` but not `Foo[5][15]`. However we can easily embed two dimensional arrays in one-dimensional ones using a one-to-one function $PAIR : \mathbb{N}^2 \to \mathbb{N}$. (For example, we can use $PAIR(x,y) = 2^x 3^y$, but there are also more efficient embeddings, see Exercise 4.1.) Hence we can replace any variable of the form `Foo[`$\langle i \rangle$`][`$\langle j \rangle$`]` with `foo[`$\langle PAIR(i,j) \rangle$`]`, and similarly for three dimensional arrays.

4.2.2 Non-Boolean variables, lists and integers

While the basic variables in NAND++ are Boolean (only have 0 or 1), we can easily extend this to other objects using encodings. For example, we can encode the alphabet {`a`,`b`,`c`,`d`,`e`,`f` } using three bits as $000, 001, 010, 011, 100, 101$. Hence, given such an encoding, we could use the code

```
Foo = REPRES("b")
```

would be a shorthand for the program

```
Foo[0]  = zero(.)
Foo[1]  = zero(.)
Foo[2]  = one(.)
```

(Where we use the constant functions `zero` and `one`, which we can apply to any variable.) Using our notion of multi-indexed arrays, we can also use code such as

```
Foo =  COPY("be")
```

as a shorthand for

```
Foo[0][0]  = zero(.)
Foo[0][1]  = one(.)
Foo[0][2]  = one(.)
Foo[1][0]  = one(.)
Foo[1][1]  = zero(.)
Foo[1][2]  = zero(.)
```

which can then in turn be mapped to standard NAND code using a one-to-one embedding $pair : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ as above.

4.2.3 Storing integers

We can also handle non-finite alphabets, such as integers, by using some prefix-free encoding and encoding the integer in an array. For example, to store non-negative integers, we can use the convention that `01` stands for 0, `11` stands for 1, and `00` is the end marker. To store integers that could be potentially negative we can use the convention `10` in the first coordinate stands for the negative sign.[3] So, code such as

```
Foo = REPRES(5)  # (1,0,1) in binary
```

will be shorthand for

```
Foo[0] = one(.)
Foo[1] = one(.)
Foo[2] = zero(.)
Foo[3] = one(.)
Foo[4] = one(.)
Foo[5] = one(.)
Foo[6] = zero(.)
Foo[7] = zero(.)
```

Using multidimensional arrays, we can use arrays of integers and hence replace code such as

```
Foo = REPRES([12,7,19,33])
```

with the equivalent NAND expressions.

[3] This is just an arbitrary choice made for concreteness, and one can choose other representations. In particular, as discussed before, if the integers are known to have a fixed size, then there is no need for additional encoding to make them prefix-free.

4.2.4 Example: Multiplying n bit numbers

We have seen in Section 4.1.6 how to use the grade-school algorithm to show that NAND programs can add n-bit numbers for every n. By following through this example, we can obtain the following result

Theorem 4.1 — Addition using NAND programs. For every n, let $ADD_n : \{0,1\}^{2n} \to \{0,1\}^{n+1}$ 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 NAND program that computes the function ADD_n. Moreover, the number of lines in this program is smaller than $100n$.

We omit the full formal proof of Theorem 4.1, but it can be obtained by going through the code in Section 4.1.6 and:

1. Proving that for every n, this code does indeed compute the addition of two n bit numbers.

2. Proving that for every n, if we expand the code out to its "unsweetened" version (i.e., to a standard NAND program), then the number of lines will be at most $100n$.

See Fig. 4.1 for a figure illustrating the number of lines our program has as a function of n. It turns out that this implementation of ADD_n uses about $13n$ lines.

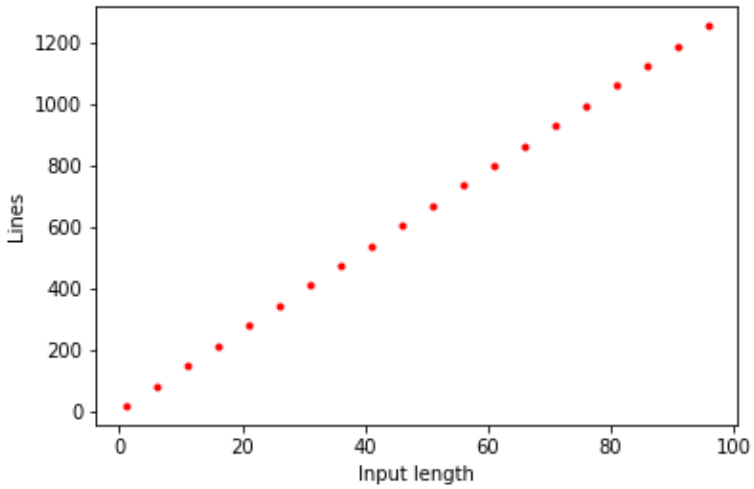


Figure 4.1: The number of lines in our NAND program to add two n bit numbers, as a function of n, for n's between 1 and 100.

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

Theorem 4.2 — Multiplication NAND programs. For every n, let $MULT_n : \{0,1\}^{2n} \to \{0,1\}^{2n}$ 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 NAND program that computes the function $MULT_n$. Moreover, the number of lines in this program is smaller than $1000n^2$.

We omit the proof, though in Exercise 4.6 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 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 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 FUNCTIONS BEYOND ARITHMETIC AND LOOKUP

We have seen that NAND 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 4.3 — Lookup function. For every k, the *lookup* function $LOOKUP_k : \{0,1\}^{2^k+k} \rightarrow \{0,1\}$ is defined as follows: For every $x \in \{0,1\}^{2^k}$ and $i \in \{0,1\}^k$,

$$LOOKUP_k(x,i) = x_i \tag{4.1}$$

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

The function $LOOKUP_1 : \{0,1\}^3 \rightarrow \{0,1\}$ 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 program. However, can we compute higher levels of $LOOKUP$? This turns out to be the case:

Theorem 4.4 — Lookup function. For every k, there is a NAND program that computes the function $LOOKUP_k : \{0,1\}^{2^k+k} \rightarrow \{0,1\}$. Moreover, the number of lines in this program is at most $4 \cdot 2^k$.

4.3.1 Constructing a NAND program for $LOOKUP$

We now prove Theorem 4.4. We will do so by induction. That is, we show how to use a NAND program for computing $LOOKUP_k$ to compute $LOOKUP_{k+1}$. Let us first see how we do this for $LOOKUP_2$. Given input $x = (x_0,x_1,x_2,x_3)$ and an index $i = (i_0,i_1)$, if the most significant bit i_1 of the index is 0 then $LOOKUP_2(x,i)$ will equal x_0 if $i_0 = 0$ and equal x_1 if $i_0 = 1$. Similarly, if the most significant bit i_1 is 1 then $LOOKUP_2(x,i)$ will equal x_2 if $i_0 = 0$ and will equal x_3 if $i_0 = 1$. Another way to say this is that

$$LOOKUP_2(x_0,x_1,x_2,x_3,i_0,i_1) = LOOKUP_1(LOOKUP_1(x_0,x_1,i_0),LOOKUP_1(x_2,x_3,i_0),i_1) \tag{4.2}$$

That is, we can compute $LOOKUP_2$ using three invocations of $LOOKUP_1$. The "pseudocode" for this program will be

```
Z[0] = LOOKUP_1(X[0],X[1],X[4])
```

```
Z[0] = LOOKUP_1(X[0],X[1],X[4])
Z[1] = LOOKUP_1(X[2],X[3],X[4])
Y[0] = LOOKUP_1(Z[0],Z[1],X[5])
```

(Note that since we call this function with $(x_0,x_1,x_2,x_3,i_0,i_1)$, the inputs `x_4` and `x_5` correspond to i_0 and i_1.) We can obtain an actual

"sugar free" NAND program of at most 12 lines by replacing the calls to `LOOKUP_1` by an appropriate copy of the program above.

We can generalize this to compute $LOOKUP_3$ using two invocations of $LOOKUP_2$ and one invocation of $LOOKUP_1$. That is, given input $x = (x_0, \ldots, x_7)$ and $i = (i_0, i_1, i_2)$ for $LOOKUP_3$, if the most significant bit of the index i_2 is 0, then the output of $LOOKUP_3$ will equal $LOOKUP_2(x_0, x_1, x_2, x_3, i_0, i_1)$, while if this index i_2 is 1 then the output will be $LOOKUP_2(x_4, x_5, x_6, x_7, i_0, i_1)$, meaning that the following pseudocode can compute $LOOKUP_3$,

```
Z[0] = LOOKUP_2(X[0],X[1],X[2],X[3],X[8],X[9])
Z[1] = LOOKUP_2(X[4],X[5],X[6],X[7],X[8],X[9])
Y[0] = LOOKUP_1(Z[0],Z[1],X[10])
```

where again we can replace the calls to `LOOKUP_2` and `LOOKUP_1` by invocations of the process above.

Formally, we can prove the following lemma:

Lemma 4.5 — 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

$$LOOKUP_1(LOOKUP_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_0, \ldots, i_{k-2}), LOOKUP_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-2}), i_{k-1}) \tag{4.3}$$

Proof. If the most significant bit i_{k-1} 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 $a = LOOKUP_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_0, \ldots, i_{k-1})$. On the other hand, if this most significant bit i_{k-1} 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 $b = LOOKUP_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-1})$. Thus we can compute $LOOKUP_k(x, i)$ by first computing a and b and then outputting $LOOKUP_1(a, b, i_{k-1})$. ■

Lemma 4.5 directly implies Theorem 4.4. We prove by induction on k that there is a NAND program of at most $4 \cdot 2^k$ lines for $LOOKUP_k$. For $k = 1$ this follows by the four line program for $LOOKUP_1$ we've seen before. For $k > 1$, we use the following pseudocode

```
a = LOOKUP_(k-1)(X[0],...,X[2^(k-1)-1],i[0],...,i[k-
 ↪  2])
b = LOOKUP_(k-1)(X[2^(k-1)],...,Z[2^(k-
 ↪  1)],i[0],...,i[k-2])
y_0 = LOOKUP_1(a,b,i[k-1])
```

In Python, this can be described as follows

```
def LOOKUP(X,i):
    k = len(i)
    if k==1: return IF(i[0],X[1],X[0])
    return IF(i[k-1],LOOKUP(X[2**(k-1):],i[:-
    ↪ 1]),LOOKUP(X[:2**(k-1)],i[:-1]))
```

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

$$L(k) \leq 2L(k-1) + 4 \,. \tag{4.4}$$

We will prove by induction that $L(k) \leq 4(2^k - 1)$. This is true for $k = 1$ by our construction. For $k > 1$, using the inductive hypothesis and Eq. (4.4), we get that

$$L(k) \leq 2 \cdot 4 \cdot (2^{k-1} - 1) + 4 = 4 \cdot 2^k - 8 + 4 = 4(2^k - 1) \tag{4.5}$$

completing the proof of Theorem 4.4. (See Fig. 4.2 for a plot of the actual number of lines in our implementation of $LOOKUP_k$.)

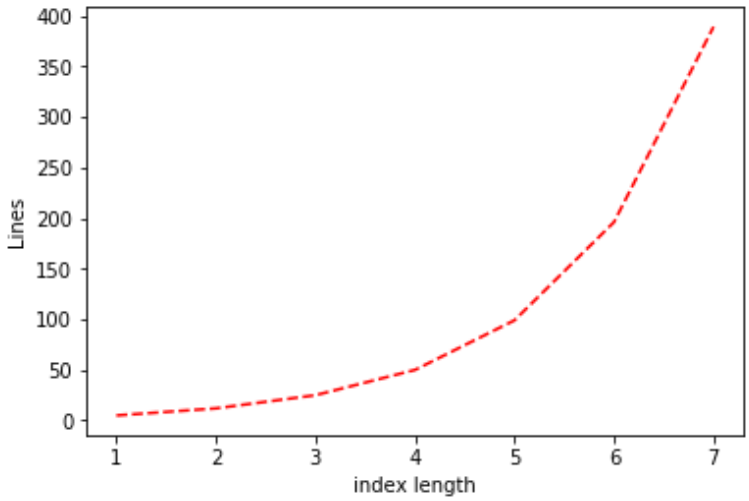


Figure 4.2: The number of lines in our implementation of the `LOOKUP_k` function as a function of k (i.e., the length of the index). The number of lines in our implementation is roughly $3 \cdot 2^k$.

4.4 COMPUTING *EVERY* FUNCTION

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

1. They can compute at least some non trivial functions.
2. Coming up with NAND 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 programs. However, it turns out we are not going to need this, as we can show in one fell swoop that NAND programs can compute *every* finite function:

Theorem 4.6 — Universality of NAND. For every n, m and function $F : \{0,1\}^n \to \{0,1\}^m$, there is a NAND program that computes the function F. Moreover, there is such a program with at most $O(m2^n)$ lines.

The implicit constant in the $O(\cdot)$ notation can be shown to be at most 10. We also note that the bound of Theorem 4.6 can be improved to $O(m2^n/n)$, see Section 4.4.2.

4.4.1 Proof of NAND's Universality

To prove Theorem 4.6, we need to give a NAND program for *every* possible function. We will restrict our attention to the case of Boolean functions (i.e., $m = 1$). In Exercise 4.8 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 $G : \{0,1\}^4 \to \{0,1\}$:[4]

Input (x)	Output ($G(x)$)
0000	1
1000	1
0100	0
1100	0
0010	1
1010	0
0110	0
1110	1
0001	0
1001	0
0101	0
1101	0
0011	1
1011	1
0111	1
1111	1

[4] In case you are curious, this is the function that computes the digits of π in the binary basis. Note that as per the convention of this course, if we think of strings as numbers then we right them with the least significant digit first.

We can see that for every $x \in \{0,1\}^4$, $G(x) = LOOKUP_4(1100100100001111, x)$. Therefore the following is NAND "pseudocode" to compute G:

```
G0000 = 1
```

```
G1000 = 1
G0100 = 0
G1100 = 0
G0010 = 1
G1010 = 0
G0110 = 0
G1110 = 1
G0001 = 0
G1001 = 0
G0101 = 0
G1101 = 0
G0011 = 1
G1011 = 1
G0111 = 1
G1111 = 1
Y[0] = LOOKUP(G0000,G1000,G0100,G1100,G0010,
              G1010,G0110,G1110,G0001,G1001,
              G0101,G1101,G0011,G1011,G1111,
              X[0],X[1],X[2],X[3])
```

Recall that we can translate this pseudocode into an actual NAND program by adding three lines to define variables `zero` and `one` that are initialized to 0 and 1 repsectively, and then replacing a statement such as `Gxxx = 0` with `Gxxx = NAND(one,one)` and a statement such as `Gxxx = 1` with `Gxxx = NAND(zero,zero)`. The call to `LOOKUP` will be replaced by the NAND program that computes $LOOKUP_4$, but we will replace the variables `X[16]`,...,`X[19]` in this program with `X[0]`,...,`X[3]` and the variables `X[0]`,...,`X[15]` with `G000`, ..., `G1111`.

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

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

2. Compute $LOOKUP_n$ on the 2^n variables initialized in the previous step, with the index variable being the input variables `X[`$\langle 0 \rangle$`]`,...,`X[`$\langle 2^n - 1 \rangle$`]`. That is, just like in the pseudocode for `G` above, we use `Y[0] = LOOKUP(F00..00,F10...00,...,F11..1,X[0],..,x[`$\langle n-1 \rangle$`])`

The total number of lines in the program will be 2^n plus the $4 \cdot 2^n$ lines that we pay for computing $LOOKUP_n$. This completes the proof of Theorem 4.6.

The NAND programming language website allows you to construct a NAND program for an arbitrary function.

R **Result in perspective** While Theorem 4.6 seems striking at first, in retrospect, it is perhaps not that surprising that every finite function can be computed with a NAND program. After all, a finite function $F : \{0,1\}^n \to \{0,1\}^m$ 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 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)

By being a little more careful, we can improve the bound of Theorem 4.6 and show that every function $F : \{0,1\}^n \to \{0,1\}^m$ can be computed by a NAND program of at most $O(m2^n/n)$ 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 $F_a : \{0,1\}^k \to \{0,1\}$ 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-k} long string P whose a^{th} entry (identifying $\{0,1\}^{n-k}$ with $[2^{n-k}]$) equals $F_a(x_{n-k}, \ldots, x_{n-1})$. One can verify that $F(x) = LOOKUP_{n-k}(P, x_0, \ldots, x_{n-k-1})$. Since we can compute $LOOKUP_{n-k}$ using $O(2^{n-k})$ lines, if we can compute the string P (i.e., compute variables `P_`$\langle 0 \rangle$, …, `P_`$\langle 2^{n-k} - 1 \rangle$) using T lines, then we can compute F in $O(2^{n-k}) + T$ lines. The trivial way to compute the string P would be to use $O(2^k)$ 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 4.6. Since there are 2^{n-k} a's, that would be a total cost of $O(2^{n-k} \cdot 2^k) = O(2^n)$ which would not improve at all on the bound of Theorem 4.6. However, a more careful observation shows that we are making some *redundant* computations. After all, there are only 2^{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_`$\langle a \rangle$ and then copy `P_`$\langle a \rangle$ to `P_`$\langle a' \rangle$ using $O(1)$ lines. Since we have 2^{2^k} unique functions, we can bound the total cost to compute P by $O(2^{2^k}2^k) + O(2^{n-k})$. Now it just becomes a matter of calculation. By our choice of k, $2^k = n - 2\log n$ and hence $2^{2^k} = \frac{2^n}{n^2}$. Since $n/2 \leq 2^k \leq n$, we can bound the total cost of computing $F(x)$ (includ-

ing also the additional $O(2^{n-k})$ cost of computing $LOOKUP_{n-k}$) by $O(\frac{2^n}{n^2} \cdot n) + O(2^n/n)$, which is what we wanted to prove.

4.4.3 The class $SIZE_{n,m}(T)$

For every $n, m, T \in \mathbb{N}$, we denote by $SIZE_{n,m}(T)$, the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$ that can be computed by NAND programs of at most T lines. Theorem 4.6 shows that $SIZE_{n,m}(4m2^n)$ is the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$. The results we've seen before can be phrased as showing that $ADD_n \in SIZE_{2n,n+1}(100n)$ and $MULT_n \in SIZE_{2n,2n}(10000n^{\log_2 3})$. See Fig. 4.3.

P Note that $SIZE_{n,m}(T)$ does *not* correspond to a set of programs! Rather, it is a set of *functions*. This distinction between *programs* and *functions* will be crucial for us in this course. You should always remember that while a program *computes* a function, it is not *equal* to a function. In particular, as we've seen, there can be more than one program to compute the same function.

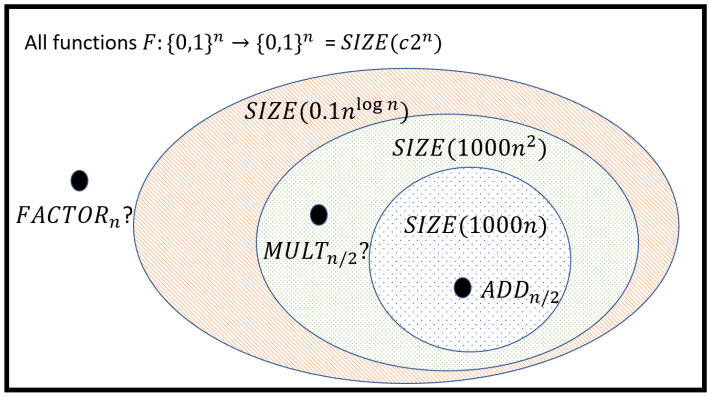


Figure 4.3: A rough illustration of the relations between the different classes of functions computed by NAND programs of given size. For every n, m, the class $SIZE_{n,m}(T)$ is a subset of the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$, and if $T \leq T$; then $SIZE_{n,m}(T) \subseteq SIZE_{n,m}(T')$. Theorem 4.6 shows that $SIZE_{n,m}(O(m \cdot 2^n))$ is equal to the set of all functions, and using Section 4.4.2 this can be improved to $O(m \cdot 2^n/n)$. If we consider all functions mapping n bits to n bits, then addition of two $n/2$ bit numbers can be done in $O(n)$ lines, while we don't know of such a program for *multiplying* two n bit numbers, though we do know it can be done in $O(n^2)$ and in fact even better size. In the above $FACTOR_n$ corresponds to the inverse problem of multiplying- finding the *prime factorization* of a given number. At the moment we do not know of any NAND program with a polynomial (or even sub-exponential) number of lines that can compute $FACTOR_n$.

R **Finite vs infinite functions** A NAND 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 program that can compute the increment function $INC : \{0,1\}^* \to \{0,1\}^*$ 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 program P_n that computes the restriction INC_n of the function 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 $10n$, $INC_n \in SIZE(10n)$ 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 $F \in SIZE(T(n))$ 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 $INC \in SIZE(10n)$. 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 $SIZE(T(n))$. 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 $f : \{0,1\}^n \to \{0,1\}$ and $s \in \mathbb{N}$, if $f \in SIZE(s)$ then $1 - f \in SIZE(s + c)$. ■

Solution: If $f \in SIZE(s)$ then there is an s-line program P that computes f. We can rename the variable `Y[0]` in P to a unique variable `unique_temp` and add the line

```
Y[0] = NAND(unique_temp,unique_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 program that computes F.
- While the NAND programming only has one operation, other operations such as functions and conditional execution can be implemented using it.
- Every function $F : \{0,1\}^n \to \{0,1\}^m$ can be computed by a NAND program of at most $O(m2^n)$ lines (and in fact at most $O(m2^n/n)$ lines).
- Sometimes (or maybe always?) we can translate an *efficient* algorithm to compute F into a NAND

program that computes F with a number of lines comparable to the number of steps in this algorithm.

4.5 EXERCISES

R **Disclaimer** Most of the exercises have been written in the summer of 2018 and haven't yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

Exercise 4.1 — Pairing. 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 program below computes the function MUX (or $LOOKUP_1$) where $MUX(a,b,c)$ equals a if $c = 0$ and equals b if $c = 1$: ■

```
t = NAND(X[2],X[2])
u = NAND(X[0],t)
v = NAND(X[1],X[2])
Y[0] = NAND(u,v)
```

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

Exercise 4.4 — Conditional statements. In this exercise we will show that even though the NAND programming language does not have an `if .. then .. else ..` statement, we can still implement it. Suppose that there is an s-line NAND program to compute $F : \{0,1\}^n \to \{0,1\}$ and an s'-line NAND program to compute $F' : \{0,1\}^n \to \{0,1\}$. Prove that there is a program of at most $s + s' + 10$ 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 — Addition. Write a program using your favorite programming language that on input an integer n, outputs a NAND program that computes ADD_n. Can you ensure that the program it outputs for ADD_n has fewer than $10n$ lines? ■

Exercise 4.6 — Multiplication. Write a program using your favorite programming language that on input an integer n, outputs a NAND program that computes $MULT_n$. Can you ensure that the program it outputs for $MULT_n$ has fewer than $1000 \cdot n^2$ lines? ■

Exercise 4.7 — Efficient multiplication (challenge). Write a program using your favorite programming language that on input an integer n, outputs a NAND program that computes $MULT_n$ and has at most $10000n^{1.9}$ lines.[5] What is the smallest number of lines you can use to multiply two 2048 bit numbers? ■

[5] **Hint:** Use Karatsuba's algorithm

Exercise 4.8 — Multibit function. Prove that

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

b. For every function $F : \{0,1\}^n \to \{0,1\}^m$, there is a NAND program of at most $10m \cdot 2^n$ lines that computes F. ■

4.6 BIBLIOGRAPHICAL NOTES

4.7 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include:

(to be completed)