Code as Data, Data as Code

Boaz Barak

**Disclaimer:** The MS-Word version of this text has significant formatting issues compared to the PDF and HTML versions.

# Code as data, data as code

### 

* See one of the most important concepts in computing: duality between code and data.
* Build up comfort in moving between different representations of programs.
* Follow the construction of a “universal circuit evaluator” that can evaluate other circuits given their representation.
* See major result that complements the result of the last chapter: some functions require an *exponential* number of gates to compute.
* Discussion of *Physical extended Church-Turing thesis* stating that Boolean circuits capture *all* feasible computation in the physical world, and its physical and philosophical implications.

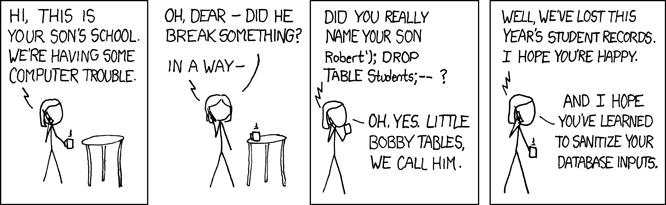
*“The term code script is, of course, too narrow. The chromosomal structures are at the same time instrumental in bringing about the development they foreshadow. They are law-code and executive power - or, to use another simile, they are architect’s plan and builder’s craft - in one.”* , Erwin Schrödinger, 1944.

*“A mathematician would hardly call a correspondence between the set of 64 triples of four units and a set of twenty other units,”universal“, while such correspondence is, probably, the most fundamental general feature of life on Earth”*, Misha Gromov, 2013

A program is simply a sequence of symbols, each of which can be encoded as a string of ’s and ’s using (for example) the ASCII standard. Therefore we can represent every NAND-CIRC program (and hence also every Boolean circuit) as a binary string. This statement seems obvious but it is actually quite profound. It means that we can treat circuits or NAND-CIRC programs both as instructions to carrying computation and also as *data* that could potentially be used as *inputs* to other computations.

A *program* is a piece of text, and so it can be fed as input to other programs.

This correspondence between *code* and *data* is one of the most fundamental aspects of computing. It underlies the notion of *general purpose* computers, that are not pre-wired to compute only one task, and also forms the basis of our hope for obtaining *general* artificial intelligence. This concept finds immense use in all areas of computing, from scripting languages to machine learning, but it is fair to say that we haven’t yet fully mastered it. Many security exploits involve cases such as “buffer overflows” when attackers manage to inject code where the system expected only “passive” data (see XKCDmomexploitsfig). The relation between code and data reaches beyond the realm of electronic computers. For example, DNA can be thought of as both a program and data (in the words of Schrödinger, who wrote before the discovery of DNA’s structure a book that inspired Watson and Crick, DNA is both “architect’s plan and builder’s craft”).

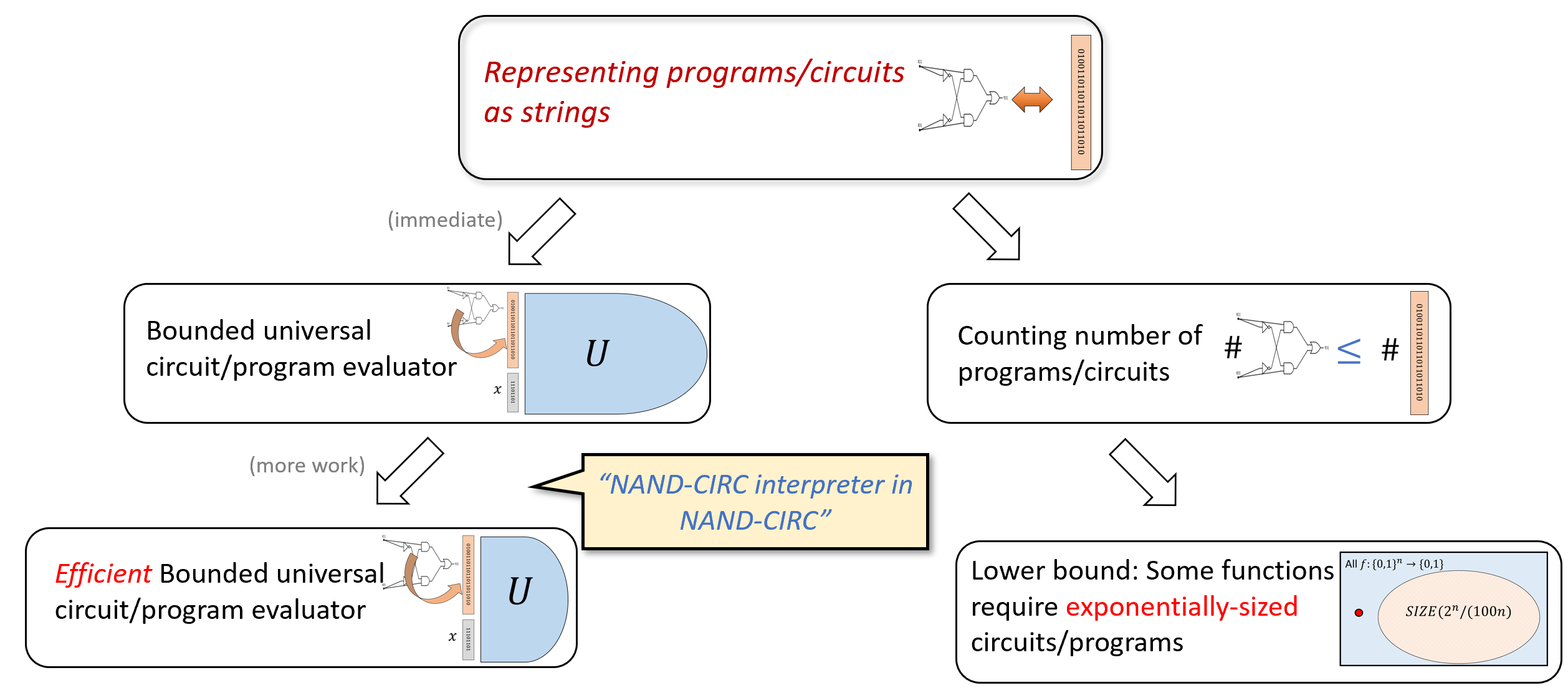


As illustrated in this xkcd cartoon, many exploits, including buffer overflow, SQL injections, and more, utilize the blurry line between “active programs” and “static strings”.

In this chapter, we will begin to explore some of the many applications of the correspondence between code and data. We start by using the representation of programs/circuits as strings to *count* the number of programs/circuits up to a certain size, and use that to obtain a counterpart to the result we proved in finiteuniversalchap. There we proved that *every* function can be computed by a circuit, but that circuit could be exponentially large (see circuit-univ-thm-improved for the precise bound) In this chapter we will prove that there are *some* functions for which we cannot do better: the *smallest* circuit that computes them is exponentially large.

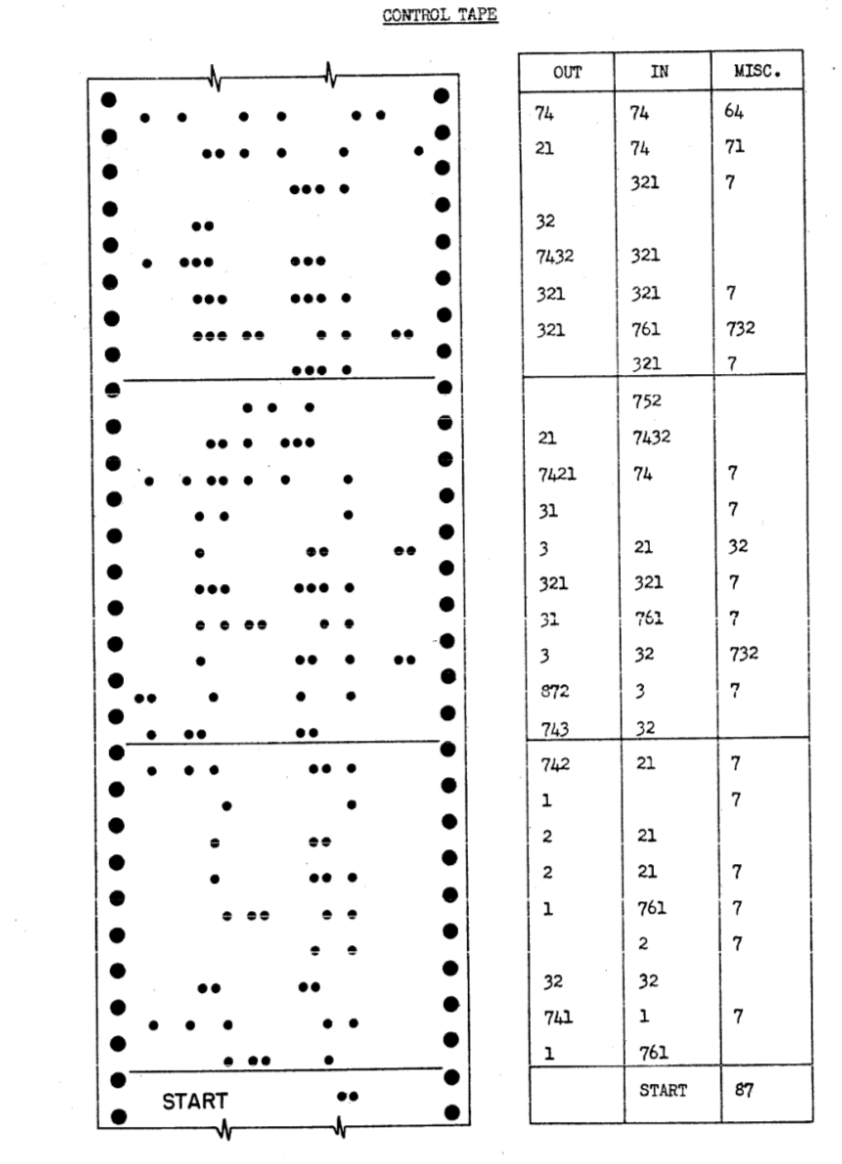
We will also use the notion of representing programs/circuits as strings to show the existence of a “universal circuit” - a circuit that can evaluate other circuits. In programming languages, this is known as a “meta circular evaluator” - a program in a certain programming language that can evaluate other programs in the same language. These results do have an important restriction: the universal circuit will have to be of bigger size than the circuits it evaluates. We will show how to get around this restriction in chaploops where we introduce *loops* and *Turing machines*.

See codedataoverviewfig for an overview of the results of this chapter.



Overview of the results in this chapter. We use the representation of programs/circuits as strings to derive two main results. First we show the existence of a universal program/circuit, and in fact (with more work) the existence of such a program/circuit whose size is at most polynomial in the size of the program/circuit it evaluates. We then use the string representation to *count* the number of programs/circuits of a given size, and use that to establish that *some* functions require an *exponential* number of lines/gates to compute.

## Representing programs as strings



In the Harvard Mark I computer, a program was represented as a list of triples of numbers, which were then encoded by perforating holes in a control card.

We can represent programs or circuits as strings in a myriad of ways. For example, since Boolean circuits are labeled directed acyclic graphs, we can use the *adjacency matrix* or *adjacency list* representations for them. However, since the code of a program is ultimately just a sequence of letters and symbols, arguably the conceptually simplest representation of a program is as such a sequence. For example, the following NAND-CIRC program

temp\_0 = NAND(X[0],X[1])  
temp\_1 = NAND(X[0],temp\_0)  
temp\_2 = NAND(X[1],temp\_0)  
Y[0] = NAND(temp\_1,temp\_2)

is simply a string of 107 symbols which include lower and upper case letters, digits, the underscore character \_ and equality sign =, punctuation marks such as “(”,“)”,“,”, spaces, and “new line” markers (often denoted as “\ n” or “↵”). Each such symbol can be encoded as a string of bits using the [ASCII](https://en.wikipedia.org/wiki/ASCII) encoding, and hence the program can be encoded as a string of length bits.

Nothing in the above discussion was specific to the program , and hence we can use the same reasoning to prove that *every* NAND-CIRC program can be represented as a string in . In fact, we can do a bit better. Since the names of the working variables of a NAND-CIRC program do not affect its functionality, we can always transform a program to have the form of where all variables apart from the inputs and outputs have the form temp\_0, temp\_1, temp\_2, etc.. Moreover, if the program has lines, then we will never need to use an index larger than (since each line involves at most three variables), and similarly the indices of the input and output variables will all be at most . Since a number between and can be expressed using at most digits, each line in the program (which has the form foo = NAND(bar,blah)), can be represented using symbols, each of which can be represented by bits. Hence an line program can be represented as a string of bits, resulting in the following theorem:

There is a constant such that for , there exists a program computing whose string representation has length at most .

We omit the formal proof of asciirepprogramthm but please make sure that you understand why it follows from the reasoning above.

## Counting programs, and lower bounds on the size of NAND-CIRC programs

One consequence of the representation of programs as strings is that the number of programs of certain length is bounded by the number of strings that represent them. This has consequences for the sets that we defined in secdefinesizeclasses.

### 

For every ,

That is, there are at most functions computed by NAND-CIRC programs of at most lines.[[1]](#footnote-35)

For any , we will show a one-to-one map from to the set of strings of length for some constant . This will conclude the proof, since it implies that is smaller than the size of the set of all strings of length at most . The size of the latter set is by the formula for sums of geometric progressions.

The map will simply map to the representation of the smallest program computing . Since , there is a program of at most lines that can be represented using a string of length at most by asciirepprogramthm. Moreover, the map is one to one, since for every distinct there must exist some input on which . This means that the programs that compute and respectively cannot be identical.

program-count has an important corollary. The number of functions that can be computed using small circuits/programs is much smaller than the total number of functions, and hence there exist functions that require very large (in fact *exponentially large*) circuits to compute. To see why this is the case, note that a function mapping to can be identified with the list of its four values on the inputs . A function mapping to can be identified with the list of its eight values on the inputs . More generally, every function can be identified with the list of its values on the inputs . Hence the number of functions mapping to is equal to the number of possible length lists of values which is exactly . Note that this is *double exponential* in , and hence even for small values of (e.g., ) the number of functions from to is truly astronomical.[[2]](#footnote-36) As mentioned, this yields the following corollary:

### 

There is a constant , such that for every sufficiently large , there is a function such that . That is, the shortest NAND-CIRC program to compute requires more than lines.[[3]](#footnote-38)

The proof is simple. If we let be the constant such that and , then setting we see that

using the fact that since , and . But since is smaller than the total number of functions mapping bits to bit, there must be at least one such function not in , which is what we needed to prove.

We have seen before that *every* function mapping to can be computed by an line program. counting-lb shows that this is tight in the sense that some functions do require such an astronomical number of lines to compute.

Some functions *cannot* be computed by a Boolean circuit using fewer than exponential (in ) number of gates.

In fact, as we explore in the exercises, this is the case for *most* functions. Hence functions that can be computed in a small number of lines (such as addition, multiplication, finding short paths in graphs, or even the function) are the exception, rather than the rule.

### 

The ASCII representation is not the shortest representation for NAND-CIRC programs. NAND-CIRC programs are equivalent to circuits with NAND gates, which means that a NAND-CIRC program of lines, inputs, and outputs can be represented by a labeled directed graph of vertices, of which have in-degree zero, and the others have in-degree at most two. Using the adjacency matrix representation for such graphs, we can reduce the implicit constant in program-count to be arbitrarily close to , see efficientrepresentationex.

### Size hierarchy theorem (optional)

By NAND-univ-thm-improved the class contains *all* functions from to , while by counting-lb, there is *some* function that is *not contained* in . In other words, for every sufficiently large ,

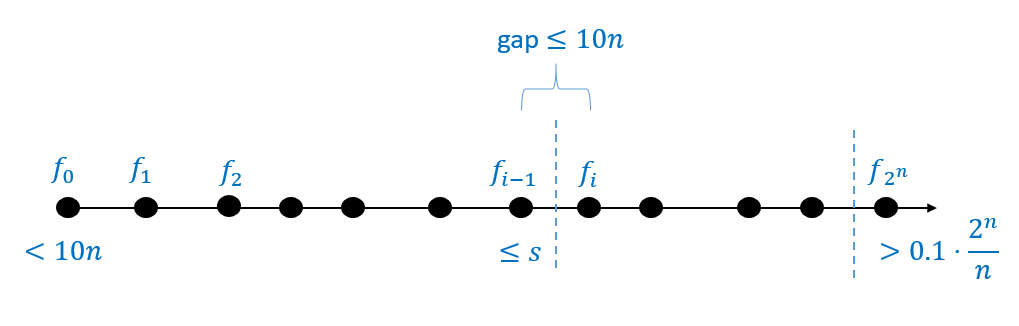
It turns out that we can use counting-lb to show a more general result: whenever we increase our “budget” of gates we can compute new functions.

### 

For every sufficiently large and ,

### 

To prove the theorem we need to find a function such that *can* be computed by a circuit of gates but it *cannot* be computed by a circuit of gates. We will do so by coming up with a sequence of functions with the following properties: **(1)** *can* be computed by a circuit of at most gates, **(2)** *cannot* be computed by a circuit of gates, and **(3)** for every , if can be computed by a circuit of size , then can be computed by a circuit of size at most . Together these properties imply that if we let be the smallest number such that , then since it must hold that which is what we need to prove. See hierarchyprooffig for an illustration.



We prove sizehiearchythm by coming up with a list of functions such that is the all zero function, is a function (obtained from counting-lb) outside of and such that and differ by one another on at most one input. We can show that for every , the number of gates to compute is at most larger than the number of gates to compute and so if we let be the smallest number such that , then .

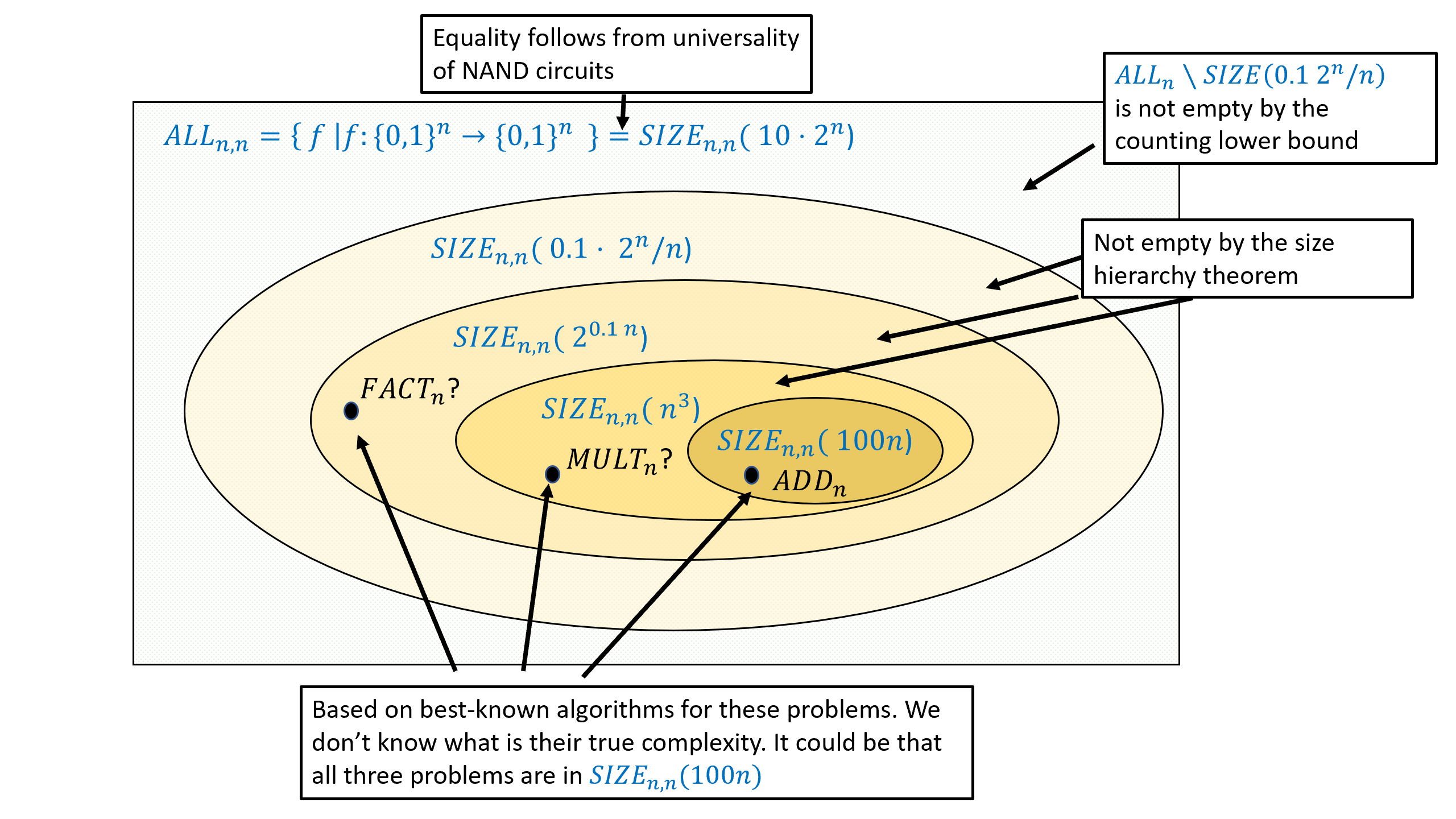
Let be the function (whose existence we are guaranteed by counting-lb) such that . We define the functions mapping to as follows. For every , if is ’s order in the lexicographical order then

The function is simply the constant zero function, while the function is equal to . Moreover, for every , the functions and differ on at most one input (i.e., the input such that ). Let , and let be the first index such that . Since there must exist such an index , and moreover since the constant zero function is a member of .

By our choice of , is a member of . To complete the proof, we need to show that . Let be the string such that and let be the value of . Then we can define also as follows

or in other words

where is the function that maps to if they are equal and to otherwise. Since (by our choice of ), can be computed using at most gates and (as can be easily verified) that , we can compute using at most gates which is what we wanted to prove.



An illustration of some of what we know about the size complexity classes (not to scale!). This figure depicts classes of the form but the state of affairs for other size complexity classes such as is similar. We know by NAND-univ-thm (with the improvement of tight-upper-bound) that all functions mapping bits to bits can be computed by a circuit of size for , while on the other hand the counting lower bound (counting-lb, see also countingmultibitex) shows that *some* such functions will require , and the size hierarchy theorem (sizehiearchythm) shows the existence of functions in whenever , see also sizehiearchyex. We also consider some specific examples: addition of two bit numbers can be done in lines, while we don’t know of such a program for *multiplying* two bit numbers, though we do know it can be done in and in fact even better size. In the above, corresponds to the inverse problem of multiplying- finding the *prime factorization* of a given number. At the moment we do not know of any circuit a polynomial (or even sub-exponential) number of lines that can compute .

While the size hierarchy theorem guarantees that there exists *some* function that *can* be computed using, for example, gates, but not using gates, we do not know of any explicit example of such a function. While we suspect that integer multiplication is such an example, we do not have any proof that this is the case.

## The tuples representation

ASCII is a fine presentation of programs, but for some applications it is useful to have a more concrete representation of NAND-CIRC programs. In this section we describe a particular choice, that will be convenient for us later on. A NAND-CIRC program is simply a sequence of lines of the form

blah = NAND(baz,boo)

There is of course nothing special about the particular names we use for variables. Although they would be harder to read, we could write all our programs using only working variables such as temp\_0, temp\_1 etc. Therefore, our representation for NAND-CIRC programs ignores the actual names of the variables, and just associate a *number* with each variable. We encode a *line* of the program as a triple of numbers. If the line has the form foo = NAND(bar,blah) then we encode it with the triple where is the number corresponding to the variable foo and and are the numbers corresponding to bar and blah respectively.

More concretely, we will associate every variable with a number in the set . The first numbers correspond to the *input* variables, the last numbers correspond to the *output* variables, and the intermediate numbers correspond to the remaining “workspace” variables. Formally, we define our representation as follows:

Let be a NAND-CIRC program of inputs, outputs, and lines, and let be the number of distinct variables used by . The *list of tuples representation of*  is the triple where is a list of triples of the form for .

We assign a number for each variable of as follows:

* For every , the variable X[] is assigned the number .
* For every , the variable Y[] is assigned the number .
* Every other variable is assigned a number in in the order in which the variable appears in the program .

The list of tuples representation is our default choice for representing NAND-CIRC programs. Since “list of tuples representation” is a bit of a mouthful, we will often call it simply “the representation” for a program . Sometimes, when the number of inputs and number of outputs are known from the context, we will simply represent a program as the list instead of the triple .

Our favorite NAND-CIRC program, the program

u = NAND(X[0],X[1])  
v = NAND(X[0],u)  
w = NAND(X[1],u)  
Y[0] = NAND(v,w)

computing the XOR function is represented as the tuple where . That is, the variables X[0] and X[1] are given the indices and respectively, the variables u,v,w are given the indices respectively, and the variable Y[0] is given the index .

Transforming a NAND-CIRC program from its representation as code to the representation as a list of tuples is a fairly straightforward programming exercise, and in particular can be done in a few lines of *Python*.[[4]](#footnote-52) The list-of-tuples representation loses information such as the particular names we used for the variables, but this is OK since these names do not make a difference to the functionality of the program.

### From tuples to strings

If is a program of size , then the number of variables is at most (as every line touches at most three variables). Hence we can encode every variable index in as a string of length , by adding leading zeroes as needed. Since this is a fixed-length encoding, it is prefix free, and so we can encode the list of triples (corresponding to the encoding of the lines of the program) as simply the string of length obtained by concatenating all of these encodings.

We define to be the length of the string representing the list corresponding to a size program. By the above we see that

We can represent as a string by prepending a prefix free representation of and to the list . Since (a program must touch at least once all its input and output variables), those prefix free representations can be encoded using strings of length . In particular, every program of at most lines can be represented by a string of length . Similarly, every circuit of at most gates can be represented by a string of length (for example by translating to the equivalent program ).

## A NAND-CIRC interpreter in NAND-CIRC

Since we can represent programs as strings, we can also think of a program as an input to a function. In particular, for every natural number we define the function as follows:

where is defined as in lengthstringrepreseq and we use the concrete representation scheme described in representprogramsec.

That is, takes as input the concatenation of two strings: a string and a string . If is a string that represents a list of triples such that is a list-of-tuples representation of a size- NAND-CIRC program , then is equal to the evaluation of the program on the input . Otherwise, equals (this case is not very important: you can simply think of as some “junk value” that indicates an error).

**Take-away points.** The fine details of ’s definition are not very crucial. Rather, what you need to remember about is that:

* is a finite function taking a string of fixed length as input and outputting a string of fixed length as output.
* is a single function, such that computing allows to evaluate *arbitrary* NAND-CIRC programs of a certain length on *arbitrary* inputs of the appropriate length.
* is a *function*, not a *program* (recall the discussion in specvsimplrem). That is, is a *specification* of what output is associated with what input. The existence of a *program* that computes (i.e., an *implementation* for ) is a separate fact, which needs to be established (and which we will do in bounded-univ, with a more efficient program shown in eff-bounded-univ).

One of the first examples of *self circularity* we will see in this book is the following theorem, which we can think of as showing a “NAND-CIRC interpreter in NAND-CIRC”:

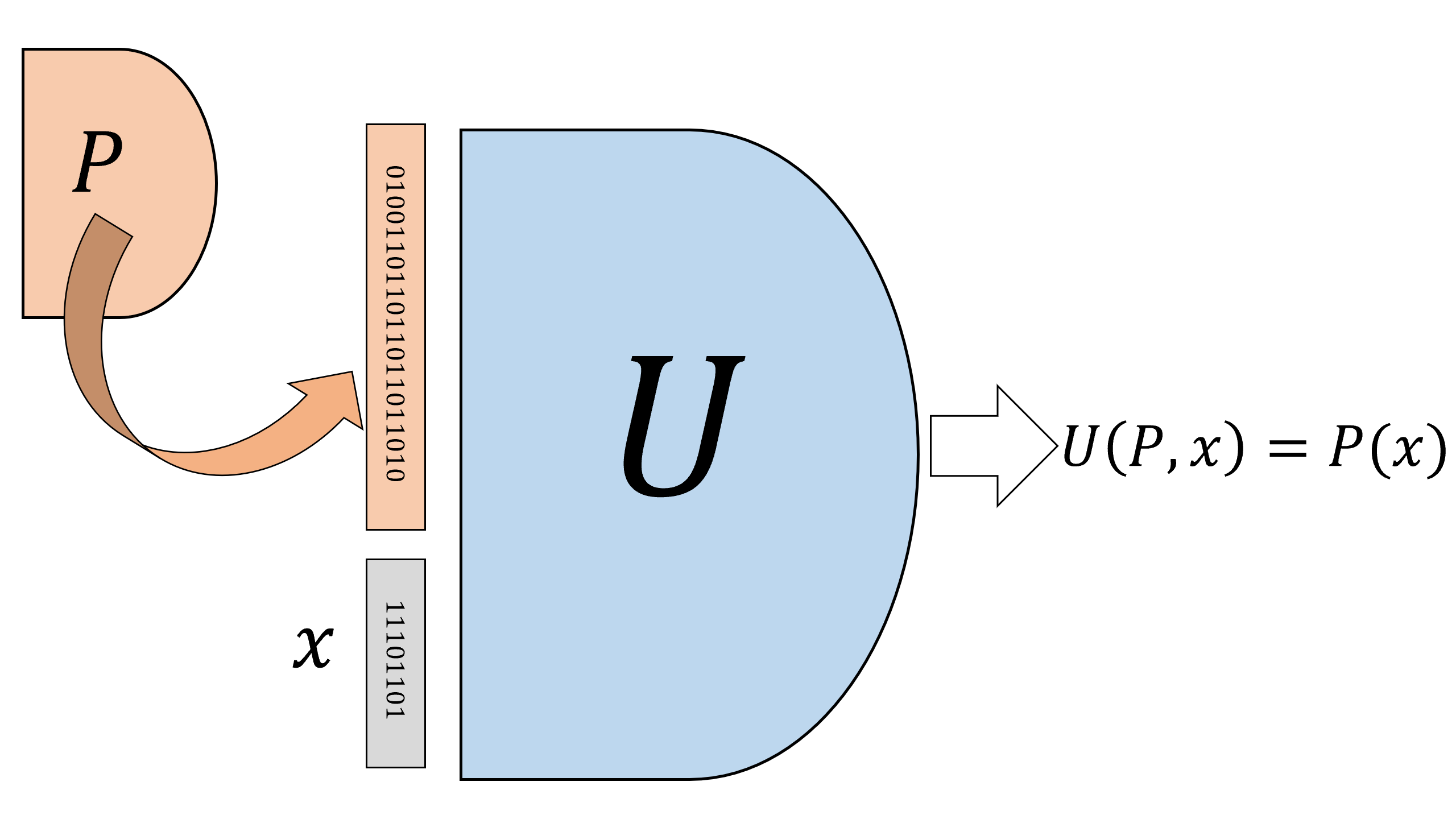
For every with there is a NAND-CIRC program that computes the function .

That is, the NAND-CIRC program takes the description of *any other NAND-CIRC program* (of the right length and inputs/outputs) and *any input* , and computes the result of evaluating the program on the input . Given the equivalence between NAND-CIRC programs and Boolean circuits, we can also think of as a circuit that takes as input the description of other circuits and their inputs, and returns their evaluation, see universalcircfig. We call this NAND-CIRC program that computes a *bounded universal program* (or a *universal circuit*, see universalcircfig). “Universal” stands for the fact that this is a *single program* that can evaluate *arbitrary* code, where “bounded” stands for the fact that only evaluates programs of bounded size. Of course this limitation is inherent for the NAND-CIRC programming language, since a program of lines (or, equivalently, a circuit of gates) can take at most inputs. Later, in chaploops, we will introduce the concept of *loops* (and the model of *Turing machines*), that allow to escape this limitation.

bounded-univ is an important result, but it is actually not hard to prove. Specifically, since is a finite function bounded-univ is an immediate corollary of NAND-univ-thm, which states that *every* finite function can be computed by *some* NAND-CIRC program.

### 

bounded-univ is simple but important. Make sure you understand what this theorem means, and why it is a corollary of NAND-univ-thm.



A *universal circuit* is a circuit that gets as input the description of an arbitrary (smaller) circuit as a binary string, and an input , and outputs the string which is the evaluation of on . We can also think of as a straight-line program that gets as input the code of a straight-line program and an input , and outputs .

### Efficient universal programs

bounded-univ establishes the existence of a NAND-CIRC program for computing , but it provides no explicit bound on the size of this program. NAND-univ-thm, which we used to prove bounded-univ, guarantees the existence of a NAND-CIRC program whose size can be as large as *exponential* in the length of its input. This would mean that even for moderately small values of (for example ), computing might require a NAND program with more lines than there are atoms in the observable universe! Fortunately, we can do much better than that. In fact, for every there exists a NAND-CIRC program for computing with size that is *polynomial* in its input length. This is shown in the following theorem.

### 

For every there is a NAND-CIRC program of at most lines that computes the function defined above (where is the number of bits needed to represent programs of lines).

If you haven’t done so already, now might be a good time to review notation in secbigohnotation. In particular, an equivalent way to state eff-bounded-univ is that it says that there *exists* some number such that *for every* , there exists a NAND-CIRC program of at most lines that computes the function .

Unlike bounded-univ, eff-bounded-univ is not a trivial corollary of the fact that every finite function can be computed by some circuit. Proving eff-bounded-univ requires us to present a concrete NAND-CIRC program for computing the function . We will do so in several stages.

1. First, we will describe the algorithm to evaluate in “pseudo code”.
2. Then, we will show how we can write a program to compute in *Python*. We will not use much about Python, and a reader that has familiarity with programming in any language should be able to follow along.
3. Finally, we will show how we can transform this Python program into a NAND-CIRC program.

This approach yields much more than just proving eff-bounded-univ: we will see that it is in fact always possible to transform (loop free) code in high level languages such as Python to NAND-CIRC programs (and hence to Boolean circuits as well).

### A NAND-CIRC interpeter in “pseudocode”

To prove eff-bounded-univ it suffices to give a NAND-CIRC program of lines that can evaluate NAND-CIRC programs of lines. Let us start by thinking how we would evaluate such programs if we weren’t restricted to only performing NAND operations. That is, let us describe informally an *algorithm* that on input , a list of triples , and a string , evaluates the program represented by on the string .

### 

It would be highly worthwhile for you to stop here and try to solve this problem yourself. For example, you can try thinking how you would write a program NANDEVAL(n,m,s,L,x) that computes this function in the programming language of your choice.

We will now describe such an algorithm. We assume that we have access to a *bit array* data structure that can store for every a bit . Specifically, if Table is a variable holding this data structure, then we assume we can perform the operations:

* GET(Table,i) which retrieves the bit corresponding to i in Table. The value of i is assumed to be an integer in .
* Table = UPDATE(Table,i,b) which updates Table so the bit corresponding to i is now set to b. The value of i is assumed to be an integer in and b is a bit in .

Input: Numbers $n,m,s$ and $t\leq 3s$, as well as a list $L$ of $s$ triples of numbers in $[t]$, and a string $x\in \{0,1\}^n$.  
  
Output: Evaluation of the program represented by $(n,m,L)$ on the -input $x\in \{0,1\}^n$.  
  
Let `Vartable` be table of size $t$  
For{$i$ in $[n]$}  
`Vartable = UPDATE(Vartable,`$i$`,`$x\_i$`)`  
Endfor  
For{$(i,j,k)$ in $L$}  
$a \leftarrow$ `GET(Vartable,`$j$`)`  
$b \leftarrow$ `GET(Vartable,`$k$`)`  
`Vartable = UPDATE(Vartable,`$i$,`NAND(`$a$`,`$b$`))`  
Endfor  
For{$j$ in $[m]$}  
$y\_j \leftarrow$ `GET(Vartable,`$t-m+j$`)`  
Endfor  
Return $y\_0,\ldots,y\_{m-1}$

evalnandcircalg evaluates the program given to it as input one line at a time, updating the Vartable table to contain the value of each variable. At the end of the execution it outputs the variables at positions which correspond to the input variables.

### A NAND interpreter in Python

To make things more concrete, let us see how we implement evalnandcircalg in the *Python* programming language. (There is nothing special about Python. We could have easily presented a corresponding function in JavaScript, C, OCaml, or any other programming language.) We will construct a function NANDEVAL that on input will output the result of evaluating the program represented by on . To keep things simple, we will not worry about the case that does not represent a valid program of inputs and outputs. The code is presented in nandevalcode.

def NANDEVAL(n,m,L,X):  
 # Evaluate a NAND-CIRC program from list of tuple representation.  
 s = len(L) # num of lines  
 t = max(max(a,b,c) for (a,b,c) in L)+1 # max index in L + 1  
 Vartable = [0] \* t # initialize array  
  
 # helper functions  
 def GET(V,i): return V[i]  
 def UPDATE(V,i,b):  
 V[i]=b  
 return V  
  
 # load input values to Vartable:  
 for i in range(n):  
 Vartable = UPDATE(Vartable,i,X[i])  
  
 # Run the program  
 for (i,j,k) in L:  
 a = GET(Vartable,j)  
 b = GET(Vartable,k)  
 c = NAND(a,b)  
 Vartable = UPDATE(Vartable,i,c)  
  
 # Return outputs Vartable[t-m], Vartable[t-m+1],....,Vartable[t-1]  
 return [GET(Vartable,t-m+j) for j in range(m)]  
  
# Test on XOR (2 inputs, 1 output)  
L = ((2, 0, 1), (3, 0, 2), (4, 1, 2), (5, 3, 4))  
print(NANDEVAL(2,1,L,(0,1))) # XOR(0,1)  
# [1]  
print(NANDEVAL(2,1,L,(1,1))) # XOR(1,1)  
# [0]

Accessing an element of the array Vartable at a given index takes a constant number of basic operations. Hence (since and ), the program above will use basic operations.[[5]](#footnote-67)

### Constructing the NAND-CIRC interpreter in NAND-CIRC

We now turn to describing the proof of eff-bounded-univ. To prove the theorem it is not enough to give a Python program. Rather, we need to show how we compute the function using a *NAND-CIRC program*. In other words, our job is to transform, for every , the Python code of #nandevalpythonsec to a NAND-CIRC program that computes the function .

### 

Before reading further, try to think how *you* could give a “constructive proof” of eff-bounded-univ. That is, think of how you would write, in the programming language of your choice, a function universal(s,n,m) that on input outputs the code for the NAND-CIRC program such that computes . There is a subtle but crucial difference between this function and the Python NANDEVAL program described above. Rather than actually evaluating a given program on some input , the function universal should output the *code* of a NAND-CIRC program that computes the map .

Our construction will follow very closely the Python implementation of EVAL above. We will use variables Vartable[],,Vartable[], where to store our variables. However, NAND doesn’t have integer-valued variables, so we cannot write code such as Vartable[i] for some variable i. However, we *can* implement the function GET(Vartable,i) that outputs the i-th bit of the array Vartable. Indeed, this is nothing but the function that we have seen in lookup-thm!

### 

Please make sure that you understand why GET and are the same function.

We saw that we can compute in time for our choice of .

For every , let correspond to the UPDATE function for arrays of length . That is, on input , , , is equal to such that

where we identify the string with a number in using the binary representation. We can compute using an line NAND-CIRC program as follows:

1. For every , there is an line NAND-CIRC program to compute the function that on input outputs if and only if is equal to (the binary representation of) . (We leave verifying this as equals and equalstwo.)
2. We have seen that we can compute the function such that equals if and if .

Together, this means that we can compute UPDATE (using some “syntactic sugar” for bounded length loops) as follows:

def UPDATE\_ell(V,i,b):  
 # Get V[0]...V[2^ell-1], i in {0,1}^ell, b in {0,1}  
 # Return NewV[0],...,NewV[2^ell-1]  
 # updated array with NewV[i]=b and all  
 # else same as V  
 for j in range(2\*\*ell): # j = 0,1,2,....,2^ell -1  
 a = EQUALS\_j(i)  
 NewV[j] = IF(a,b,V[j])  
 return NewV

Since the loop over j in UPDATE is run times, and computing EQUALS\_j takes lines, the total number of lines to compute UPDATE is . Once we can compute GET and UPDATE, the rest of the implementation amounts to “book keeping” that needs to be done carefully, but is not too insightful, and hence we omit the full details. Since we run GET and UPDATE times, the total number of lines for computing is . This completes (up to the omitted details) the proof of eff-bounded-univ.

The NAND-CIRC program above is less efficient than its Python counterpart, since NAND does not offer arrays with efficient random access. Hence for example the LOOKUP operation on an array of bits takes lines in NAND even though it takes steps (or maybe steps, depending on how we count) in *Python*.

It turns out that it is possible to improve the bound of eff-bounded-univ, and evaluate line NAND-CIRC programs using a NAND-CIRC program of lines. The key is to consider the description of NAND-CIRC programs as circuits, and in particular as directed acyclic graphs (DAGs) of bounded in-degree. A universal NAND-CIRC program for line programs will correspond to a *universal graph* for such vertex DAGs. We can think of such a graph as fixed “wiring” for a communication network, that should be able to accommodate any arbitrary pattern of communication between vertices (where this pattern corresponds to an line NAND-CIRC program). It turns out that such efficient [routing networks](https://goo.gl/NnkkjM) exist that allow embedding any vertex circuit inside a universal graph of size , see the bibliographical notes bibnotescodeasdata for more on this issue.

## A Python interpreter in NAND-CIRC (discussion)

To prove eff-bounded-univ we essentially translated every line of the Python program for EVAL into an equivalent NAND-CIRC snippet. However, none of our reasoning was specific to the particular function . It is possible to translate *every* Python program into an equivalent NAND-CIRC program of comparable efficiency. (More concretely, if the Python program takes operations on inputs of length at most then there exists NAND-CIRC program of lines that agrees with the Python program on inputs of length .) Actually doing so requires taking care of many details and is beyond the scope of this book, but let me try to convince you why you should believe it is possible in principle.

For starters, one can use [CPython](https://en.wikipedia.org/wiki/CPython) (the reference implementation for Python), to evaluate every Python program using a C program. We can combine this with a C compiler to transform a Python program to various flavors of “machine language”. So, to transform a Python program into an equivalent NAND-CIRC program, it is enough to show how to transform a *machine language* program into an equivalent NAND-CIRC program. One minimalistic (and hence convenient) family of machine languages is known as the *ARM architecture* which powers many mobile devices including essentially all Android devices.[[6]](#footnote-75) There are even simpler machine languages, such as the [LEG architecture](https://github.com/frasercrmck/llvm-leg) for which a backend for the [LLVM compiler](http://llvm.org/) was implemented (and hence can be the target of compiling any of the [large and growing list](https://en.wikipedia.org/wiki/LLVM#Front_ends) of languages that this compiler supports). Other examples include the [TinyRAM](http://www.scipr-lab.org/doc/TinyRAM-spec-0.991.pdf) architecture (motivated by interactive proof systems that we will discuss in chapproofs) and the teaching-oriented [Ridiculously Simple Computer](https://www.ece.umd.edu/~blj/RiSC/) architecture. Going one by one over the instruction sets of such computers and translating them to NAND snippets is no fun, but it is a feasible thing to do. In fact, ultimately this is very similar to the transformation that takes place in converting our high level code to actual silicon gates that are not so different from the operations of a NAND-CIRC program. Indeed, tools such as [MyHDL](http://www.myhdl.org/) that transform “Python to Silicon” can be used to convert a Python program to a NAND-CIRC program.

The NAND-CIRC programming language is just a teaching tool, and by no means do I suggest that writing NAND-CIRC programs, or compilers to NAND-CIRC, is a practical, useful, or enjoyable activity. What I do want is to make sure you understand why it *can* be done, and to have the confidence that if your life (or at least your grade) depended on it, then you would be able to do this. Understanding how programs in high level languages such as Python are eventually transformed into concrete low-level representation such as NAND is fundamental to computer science.

The astute reader might notice that the above paragraphs only outlined why it should be possible to find for every *particular* Python-computable function , a *particular* comparably efficient NAND-CIRC program that computes . But this still seems to fall short of our goal of writing a “Python interpreter in NAND” which would mean that for every parameter , we come up with a *single* NAND-CIRC program such that given a description of a Python program , a particular input , and a bound on the number of operations (where the lengths of and and the value of are all at most ) returns the result of executing on for at most steps. After all, the transformation above takes every Python program into a *different* NAND-CIRC program, and so does not yield “one NAND-CIRC program to rule them all” that can evaluate every Python program up to some given complexity. However, we can in fact obtain one NAND-CIRC program to evaluate *arbitrary* Python programs. The reason is that there exists a Python interpreter *in Python*: a Python program that takes a bit string, interprets it as Python code, and then runs that code. Hence, we only need to show a NAND-CIRC program that computes the same function as the particular Python program , and this will give us a way to evaluate *all* Python programs.

What we are seeing time and again is the notion of *universality* or *self reference* of computation, which is the sense that all reasonably rich models of computation are expressive enough that they can “simulate themselves”. The importance of this phenomenon to both the theory and practice of computing, as well as far beyond it, including the foundations of mathematics and basic questions in science, cannot be overstated.

## The physical extended Church-Turing thesis (discussion)

We’ve seen that NAND gates (and other Boolean operations) can be implemented using very different systems in the physical world. What about the reverse direction? Can NAND-CIRC programs simulate any physical computer?

We can take a leap of faith and stipulate that Boolean circuits (or equivalently NAND-CIRC programs) do actually encapsulate *every* computation that we can think of. Such a statement (in the realm of infinite functions, which we’ll encounter in chaploops) is typically attributed to Alonzo Church and Alan Turing, and in that context is known as the *Church-Turing Thesis*. As we will discuss in future lectures, the Church-Turing Thesis is not a mathematical theorem or conjecture. Rather, like theories in physics, the Church-Turing Thesis is about mathematically modeling the real world. In the context of finite functions, we can make the following informal hypothesis or prediction:

**“Physical Extended Church-Turing Thesis” (PECTT):** *If a function can be computed in the physical world using amount of “physical resources” then it can be computed by a Boolean circuit program of roughly gates.*

A priori it might seem rather extreme to hypothesize that our meager model of NAND-CIRC programs or Boolean circuits captures all possible physical computation. But yet, in more than a century of computing technologies, no one has yet built any scalable computing device that challenges this hypothesis.

We now discuss the “fine print” of the PECTT in more detail, as well as the (so far unsuccessful) challenges that have been raised against it. There is no single universally-agreed-upon formalization of “roughly physical resources”, but we can approximate this notion by considering the size of any physical computing device and the time it takes to compute the output, and ask that any such device can be simulated by a Boolean circuit with a number of gates that is a polynomial (with not too large exponent) in the size of the system and the time it takes it to operate.

In other words, we can phrase the PECTT as stipulating that any function that can be computed by a device that takes a certain volume of space and requires time to complete the computation, must be computable by a Boolean circuit with a number of gates that is polynomial in and .

The exact form of the function is not universally agreed upon but it is generally accepted that if is an *exponentially hard* function, in the sense that it has no NAND-CIRC program of fewer than, say, lines, then a demonstration of a physical device that can compute in the real world for moderate input lengths (e.g., ) would be a violation of the PECTT.

We can attempt a more exact phrasing of the PECTT as follows. Suppose that is a physical system that accepts binary stimuli and has a binary output, and can be enclosed in a sphere of volume . We say that the system *computes* a function within seconds if whenever we set the stimuli to some value , if we measure the output after seconds then we obtain .

One can then phrase the PECTT as stipulating that if there exists such a system that computes within seconds, then there exists a NAND-CIRC program that computes and has at most lines, where is some normalization constant. (We can also consider variants where we use [surface area](https://goo.gl/ALgbVS) instead of volume, or take to a different power than . However, none of these choices makes a qualitative difference to the discussion below.) In particular, suppose that is a function that requires lines for any NAND-CIRC program (such a function exists by counting-lb). Then the PECTT would imply that either the volume or the time of a system that computes will have to be at least . Since this quantity grows *exponentially* in , it is not hard to set parameters so that even for moderately large values of , such a system could not fit in our universe.

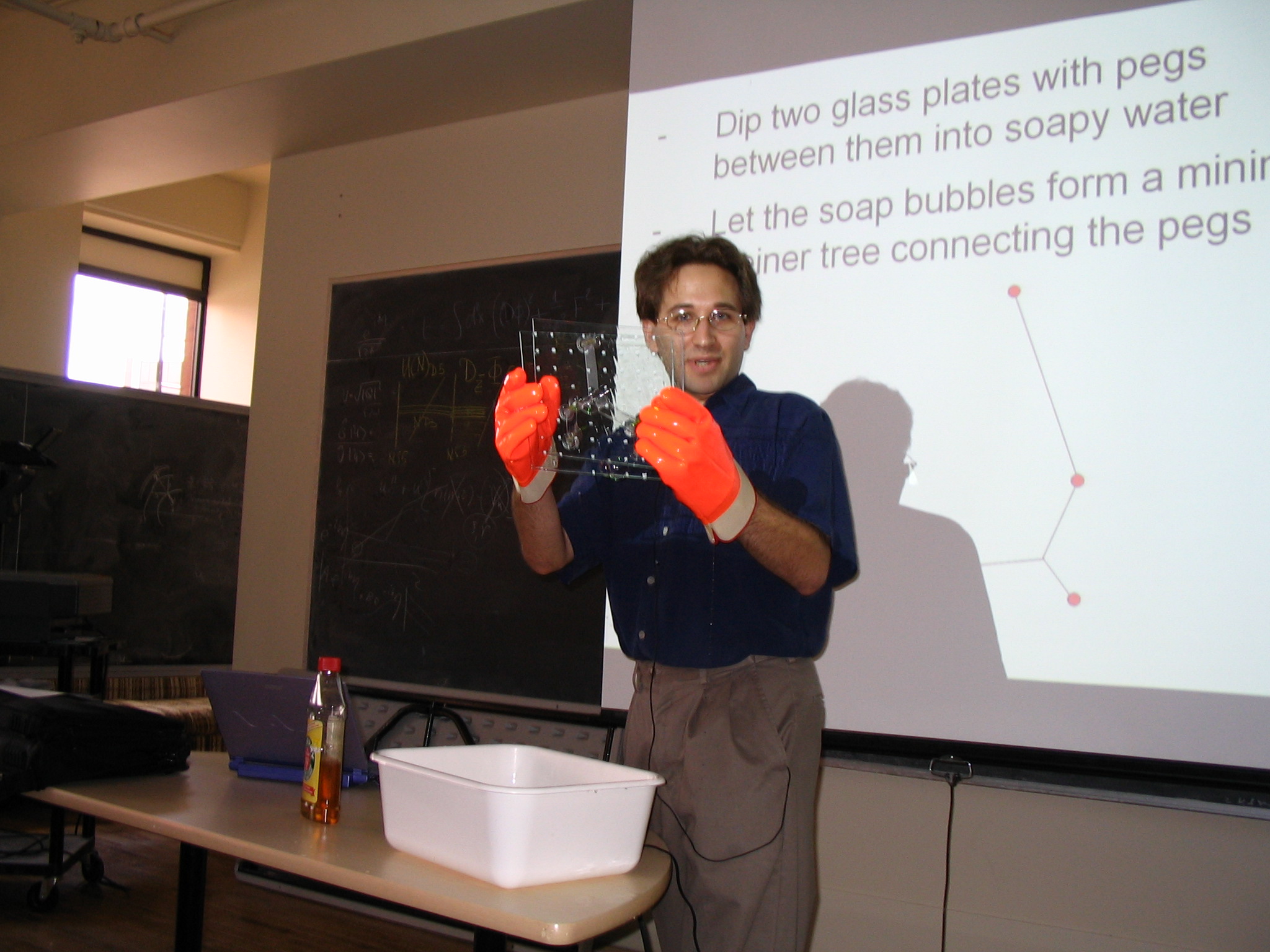
To fully make the PECTT concrete, we need to decide on the units for measuring time and volume, and the normalization constant . One conservative choice is to assume that we could squeeze computation to the absolute physical limits (which are many orders of magnitude beyond current technology). This corresponds to setting and using the [Planck units](https://goo.gl/gkpmBF) for volume and time. The *Planck length* (which is, roughly speaking, the shortest distance that can theoretically be measured) is roughly meters. The *Planck time* (which is the time it takes for light to travel one Planck length) is about seconds. In the above setting, if a function takes, say, 1KB of input (e.g., roughly bits, which can encode a by bitmap image), and requires at least NAND lines to compute, then any physical system that computes it would require either volume of Planck length cubed, which is more than meters cubed or take at least Planck Time units, which is larger than seconds. To get a sense of how big that number is, note that the universe is only about seconds old, and its observable radius is only roughly meters. The above discussion suggests that it is possible to *empirically falsify* the PECTT by presenting a smaller-than-universe-size system that computes such a function.

There are of course several hurdles to refuting the PECTT in this way, one of which is that we can’t actually test the system on all possible inputs. However, it turns out that we can get around this issue using notions such as *interactive proofs* and *program checking* that we might encounter later in this book. Another, perhaps more salient problem, is that while we know many hard functions exist, at the moment there is *no single explicit function* for which we can *prove* an (let alone ) lower bound on the number of lines that a NAND-CIRC program needs to compute it.

### Attempts at refuting the PECTT

One of the admirable traits of mankind is the refusal to accept limitations. In the best case this is manifested by people achieving longstanding “impossible” challenges such as heavier-than-air flight, putting a person on the moon, circumnavigating the globe, or even resolving [Fermat’s Last Theorem](https://en.wikipedia.org/wiki/Fermat%27s_Last_Theorem). In the worst case it is manifested by people continually following the footsteps of previous failures to try to do proven-impossible tasks such as build a [perpetual motion machine](https://en.wikipedia.org/wiki/Perpetual_motion), [trisect an angle](https://en.wikipedia.org/wiki/Angle_trisection) with a compass and straightedge, or refute [Bell’s inequality](https://en.wikipedia.org/wiki/Bell%27s_theorem). The Physical Extended Church-Turing thesis (in its various forms) has attracted both types of people. Here are some physical devices that have been speculated to achieve computational tasks that cannot be done by not-too-large NAND-CIRC programs:

* **Spaghetti sort:** One of the first lower bounds that Computer Science students encounter is that sorting numbers requires making comparisons. The “spaghetti sort” is a description of a proposed “mechanical computer” that would do this faster. The idea is that to sort numbers , we could cut spaghetti noodles into lengths , and then if we simply hold them together in our hand and bring them down to a flat surface, they will emerge in sorted order. There are a great many reasons why this is not truly a challenge to the PECTT hypothesis, and I will not ruin the reader’s fun in finding them out by her or himself.
* **Soap bubbles:** One function that is conjectured to require a large number of NAND lines to solve is the *Euclidean Steiner Tree* problem. This is the problem where one is given points in the plane (say with integer coordinates ranging from till , and hence the list can be represented as a string of size) and some number . The goal is to figure out whether it is possible to connect all the points by line segments of total length at most . This function is conjectured to be hard because it is *NP complete* - a concept that we’ll encounter later in this course - and it is in fact reasonable to conjecture that as grows, the number of NAND lines required to compute this function grows *exponentially* in , meaning that the PECTT would predict that if is sufficiently large (such as few hundreds or so) then no physical device could compute . Yet, some people claimed that there is in fact a very simple physical device that could solve this problem, that can be constructed using some wooden pegs and soap. The idea is that if we take two glass plates, and put wooden pegs between them in the locations then bubbles will form whose edges touch those pegs in a way that will minimize the total energy which turns out to be a function of the total length of the line segments. The problem with this device is that nature, just like people, often gets stuck in “local optima”. That is, the resulting configuration will not be one that achieves the absolute minimum of the total energy but rather one that can’t be improved with local changes. [Aaronson](http://www.scottaaronson.com/papers/npcomplete.pdf) has carried out actual experiments (see aaronsonsoapfig), and saw that while this device often is successful for three or four pegs, it starts yielding suboptimal results once the number of pegs grows beyond that.



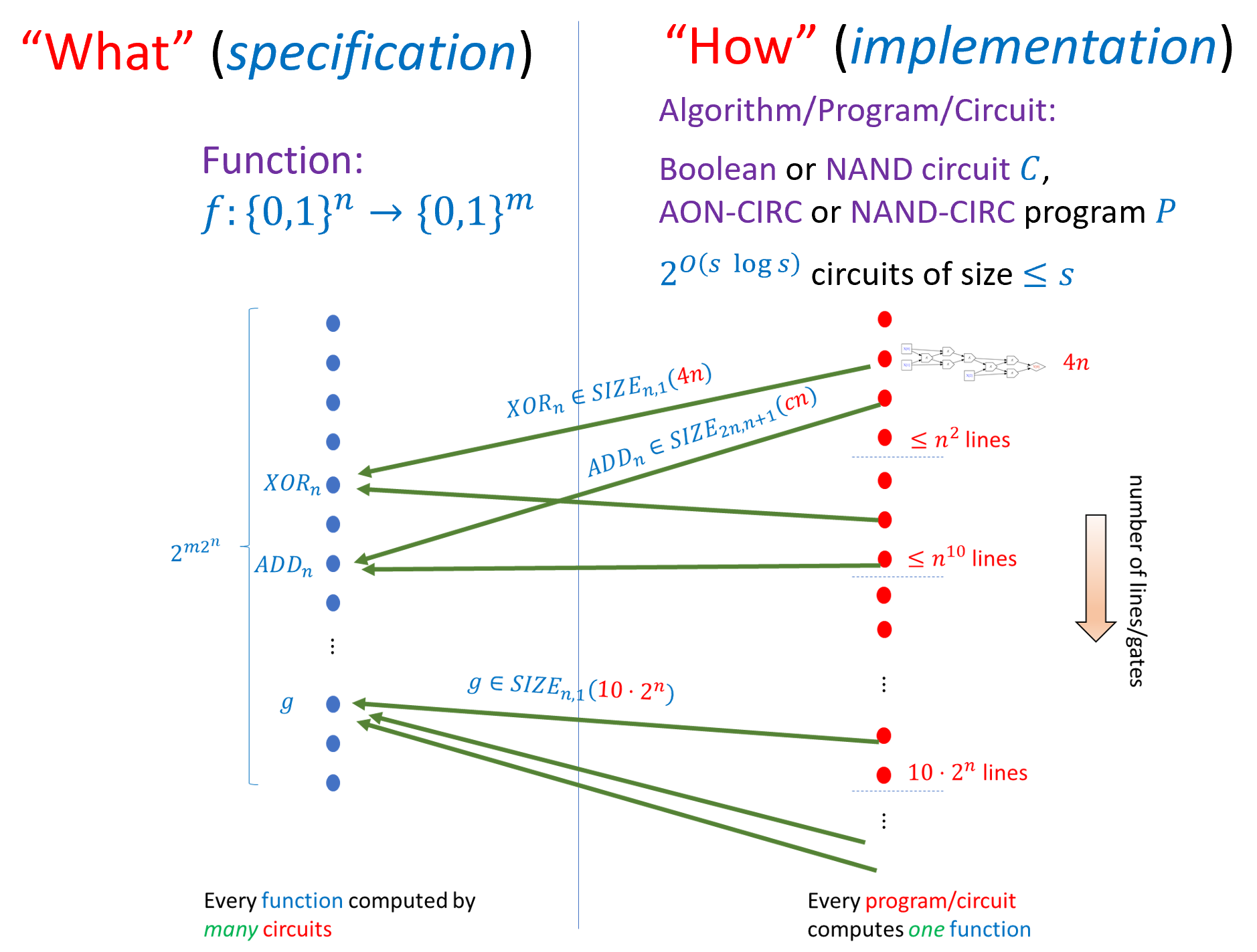
Scott Aaronson [tests](http://www.scottaaronson.com/blog/?p=266) a candidate device for computing Steiner trees using soap bubbles.

* **DNA computing.** People have suggested using the properties of DNA to do hard computational problems. The main advantage of DNA is the ability to potentially encode a lot of information in a relatively small physical space, as well as compute on this information in a highly parallel manner. At the time of this writing, it was [demonstrated](http://science.sciencemag.org/content/337/6102/1628.full) that one can use DNA to store about bits of information in a region of radius about a millimeter, as opposed to about bits with the best known hard disk technology. This does not posit a real challenge to the PECTT but does suggest that one should be conservative about the choice of constant and not assume that current hard disk + silicon technologies are the absolute best possible.[[7]](#footnote-96)
* **Continuous/real computers.** The physical world is often described using continuous quantities such as time and space, and people have suggested that analog devices might have direct access to computing with real-valued quantities and would be inherently more powerful than discrete models such as NAND machines. Whether the “true” physical world is continuous or discrete is an open question. In fact, we do not even know how to precisely *phrase* this question, let alone answer it. Yet, regardless of the answer, it seems clear that the effort to measure a continuous quantity grows with the level of accuracy desired, and so there is no “free lunch” or way to bypass the PECTT using such machines (see also [this paper](http://www.cs.princeton.edu/~ken/MCS86.pdf)). Related to that are proposals known as “hypercomputing” or “Zeno’s computers” which attempt to use the continuity of time by doing the first operation in one second, the second one in half a second, the third operation in a quarter second and so on.. These fail for a similar reason to the one guaranteeing that Achilles will eventually catch the tortoise despite the original Zeno’s paradox.
* **Relativity computer and time travel.** The formulation above assumed the notion of time, but under the theory of relativity time is in the eye of the observer. One approach to solve hard problems is to leave the computer to run for a lot of time from *his* perspective, but to ensure that this is actually a short while from *our* perspective. One approach to do so is for the user to start the computer and then go for a quick jog at close to the speed of light before checking on its status. Depending on how fast one goes, few seconds from the point of view of the user might correspond to centuries in computer time (it might even finish updating its Windows operating system!). Of course the catch here is that the energy required from the user is proportional to how close one needs to get to the speed of light. A more interesting proposal is to use time travel via *closed timelike curves (CTCs)*. In this case we could run an arbitrarily long computation by doing some calculations, remembering the current state, and then travelling back in time to continue where we left off. Indeed, if CTCs exist then we’d probably have to revise the PECTT (though in this case I will simply travel back in time and edit these notes, so I can claim I never conjectured it in the first place…)
* **Humans.** Another computing system that has been proposed as a counterexample to the PECTT is a 3 pound computer of about 0.1m radius, namely the human brain. Humans can walk around, talk, feel, and do other things that are not commonly done by NAND-CIRC programs, but can they compute partial functions that NAND-CIRC programs cannot? There are certainly computational tasks that *at the moment* humans do better than computers (e.g., play some [video games](http://www.theverge.com/2016/11/4/13518210/deepmind-starcraft-ai-google-blizzard), at the moment), but based on our current understanding of the brain, humans (or other animals) have no *inherent* computational advantage over computers. The brain has about neurons, each operating at a speed of about operations per seconds. Hence a rough first approximation is that a Boolean circuit of about gates could simulate one second of a brain’s activity.[[8]](#footnote-99) Note that the fact that such a circuit (likely) exists does not mean it is easy to *find* it. After all, constructing this circuit took evolution billions of years. Much of the recent efforts in artificial intelligence research is focused on finding programs that replicate some of the brain’s capabilities and they take massive computational effort to discover, these programs often turn out to be much smaller than the pessimistic estimates above. For example, at the time of this writing, Google’s [neural network for machine translation](https://arxiv.org/pdf/1609.08144.pdf) has about nodes (and can be simulated by a NAND-CIRC program of comparable size). Philosophers, priests and many others have since time immemorial argued that there is something about humans that cannot be captured by mechanical devices such as computers; whether or not that is the case, the evidence is thin that humans can perform computational tasks that are inherently impossible to achieve by computers of similar complexity.[[9]](#footnote-101)
* **Quantum computation.** The most compelling attack on the Physical Extended Church-Turing Thesis comes from the notion of *quantum computing*. The idea was initiated by the observation that systems with strong quantum effects are very hard to simulate on a computer. Turning this observation on its head, people have proposed using such systems to perform computations that we do not know how to do otherwise. At the time of this writing, scalable quantum computers have not yet been built, but it is a fascinating possibility, and one that does not seem to contradict any known law of nature. We will discuss quantum computing in much more detail in quantumchap. Modeling quantum computation involves extending the model of Boolean circuits into *Quantum circuits* that have one more (very special) gate. However, the main takeaway is that while quantum computing does suggest we need to amend the PECTT, it does *not* require a complete revision of our worldview. Indeed, almost all of the content of this book remains the same regardless of whether the underlying computational model is Boolean circuits or quantum circuits.

While even the precise phrasing of the PECTT, let alone understanding its correctness, is still a subject of active research, some variants of it are already implicitly assumed in practice. Governments, companies, and individuals currently rely on *cryptography* to protect some of their most precious assets, including state secrets, control of weapon systems and critical infrastructure, securing commerce, and protecting the confidentiality of personal information. In applied cryptography, one often encounters statements such as “cryptosystem provides 128 bits of security”. What such a statement really means is that **(a)** it is conjectured that there is no Boolean circuit (or, equivalently, a NAND-CIRC program) of size much smaller than that can break , and **(b)** we assume that no other physical mechanism can do better, and hence it would take roughly a amount of “resources” to break . We say “conjectured” and not “proved” because, while we can phrase the statement that breaking the system cannot be done by an -gate circuit as a precise mathematical conjecture, at the moment we are unable to *prove* such a statement for any non-trivial cryptosystem. This is related to the vs question we will discuss in future chapters. We will explore Cryptography in chapcryptography.

### 

* We can think of programs both as describing a *process*, as well as simply a list of symbols that can be considered as *data* that can be fed as input to other programs.
* We can write a NAND-CIRC program that evaluates arbitrary NAND-CIRC programs (or equivalently a circuit that evaluates other circuits). Moreover, the efficiency loss in doing so is not too large.
* We can even write a NAND-CIRC program that evaluates programs in other programming languages such as Python, C, Lisp, Java, Go, etc.
* By a leap of faith, we could hypothesize that the number of gates in the smallest circuit that computes a function captures roughly the amount of physical resources required to compute . This statement is known as the *Physical Extended Church-Turing Thesis (PECTT)*.
* Boolean circuits (or equivalently AON-CIRC or NAND-CIRC programs) capture a surprisingly wide array of computational models. The strongest currently known challenge to the PECTT comes from the potential for using quantum mechanical effects to speed-up computation, a model known as *quantum computers*.



A finite computational task is specified by a function . We can model a computational process using Boolean circuits (of varying gate sets) or straight-line program. Every function can be computed by many programs. We say that if there exists a NAND circuit of at most gates (equivalently a NAND-CIRC program of at most lines) that computes . Every function can be computed by a circuit of gates. Many functions such as multiplication, addition, solving linear equations, computing the shortest path in a graph, and others, can be computed by circuits of much fewer gates. In particular there is an -size circuit that computes the map where is a string describing a circuit of gates. However, the counting argument shows there do exist some functions that require gates to compute.

## Recap of Part I: Finite Computation

This chapter concludes the first part of this book that deals with *finite computation* (computing functions that map a fixed number of Boolean inputs to a fixed number of Boolean outputs). The main take-aways from compchap, finiteuniversalchap, and codeanddatachap are as follows (see also finiterecapfig):

* We can formally define the notion of a function being computable using basic operations. Whether these operations are AND/OR/NOT, NAND, or some other universal basis does not make much difference. We can describe such a computation either using a *circuit* or using a *straight-line program*.
* We define to be the set of *functions* that are computable by NAND circuits of at most gates. This set is equal to the set of functions computable by a NAND-CIRC program of at most lines and up to a constant factor in (which we will not care about); this is also the same as the set of functions that are computable by a Boolean circuit of at most AND/OR/NOT gates. The class is a set of *functions*, not of programs/circuits.
* *Every* function can be computed using a circuit of *at most* gates. *Some* functions require *at least* gates. We define to be the set of functions from to that can be computed using at most gates.
* We can describe a circuit/program as a string. For every , there is a *universal* circuit/program that can evaluate programs of length given their description as strings. We can use this representation also to *count* the number of circuits of at most gates and hence prove that some functions cannot be computed by circuits of smaller-than-exponential size.
* If there is a circuit of gates that computes a function , then we can build a physical device to compute using basic components (such as transistors). The “Physical Extended Church-Turing Thesis” postulates that the reverse direction is true as well: if is a function for which *every* circuit requires at least gates then that *every* physical device to compute will require about “physical resources”. The main challenge to the PECTT is *quantum computing*, which we will discuss in quantumchap.

**Sneak preview:** In the next part we will discuss how to model computational tasks on *unbounded inputs*, which are specified using functions (or ) that can take an unbounded number of Boolean inputs.

## Exercises

Which one of the following statements is false:

1. There is an line NAND-CIRC program that given as input program of lines in the list-of-tuples representation computes the output of when all its input are equal to .
2. There is an line NAND-CIRC program that given as input program of characters encoded as a string of bits using the ASCII encoding, computes the output of when all its input are equal to .
3. There is an line NAND-CIRC program that given as input program of lines in the list-of-tuples representation computes the output of when all its input are equal to .

### 

For every , show that there is an line NAND-CIRC program that computes the function where if and only if .

### 

For every and , show that there is an line NAND-CIRC program that computes the function that on input outputs if and only if .

Prove that there exists a number such that for every sufficiently large and every there exists a function that requires at least NAND gates to compute. See footnote for hint.[[10]](#footnote-113)

Prove that there exists a number such that for every and there exists a function . See footnote for hint.[[11]](#footnote-115)

Use the ideas of efficientrepresentation to show that for every and sufficiently large ,

Conclude that the implicit constant in program-count can be made arbitrarily close to . See footnote for hint.[[12]](#footnote-117)

Prove that for every , if is sufficiently large then there exists a function such that . See footnote for hint.[[13]](#footnote-119)

### 

Suppose and that we choose a function at random, choosing for every the value to be the result of tossing an independent unbiased coin. Prove that the probability that there is a line program that computes is at most .[[14]](#footnote-122)

The following is a tuple representing a NAND program: .

1. Write a table with the eight values , , , , , , , in this order.
2. Describe what the programs does in words.

For every sufficiently large , let be the function that takes an -length string that encodes a pair where and is a NAND program of inputs, a single output, and at most lines, and returns the output of on .[[15]](#footnote-123) That is, .

Prove that for every sufficiently large , there *does not exist* an XOR circuit that computes the function , where a XOR circuit has the gate as well as the constants and (see xorex). That is, prove that there is some constant such that for every and XOR circuit of inputs and a single output, there exists a pair such that .

(This exercise assumes background in probability theory and/or machine learning that you might not have at this point. Feel free to come back to it at a later point and in particular after going over probabilitychap.) In this exercise we will use our bound on the number of circuits of size to show that (if we ignore the cost of computation) every such circuit can be *learned* from not too many training samples. Specifically, if we find a size- circuit that classifies correctly a training set of samples from some distribution , then it is guaranteed to do well on the whole distribution . Since Boolean circuits model very many physical processes (maybe even all of them, if the (controversial) physical extended Church-Turing thesis is true), this shows that all such processes could be learned as well (again, ignoring the computation cost of finding a classifier that does well on the training data).

Let be any probability distribution over and let be a NAND circuit with inputs, one output, and size . Prove that there is some constant such that with probability at least the following holds: if and are chosen independently from , then for every circuit such that on every , .

In other words, if is a so called “empirical risk minimizer” that agrees with on all the training examples , then it will also agree with with high probability for samples drawn from the distribution (i.e., it “generalizes”, to use Machine-Learning lingo). See footnote for hint.[[16]](#footnote-125)

## Bibliographical notes

The function is usually known as a *universal circuit*. The implementation we describe is not the most efficient known. Valiant [@Valiant1976] first showed a universal circuit of size where is the size of the input. Universal circuits have seen in recent years new motivations due to their applications for cryptography, see [@lipmaa2016valiant, @Gunther2017] .

While we’ve seen that “most” functions mapping bits to one bit require circuits of exponential size , we actually do not know of any *explicit* function for which we can *prove* that it requires, say, at least or even size. At the moment, the strongest such lower bound we know is that there are quite simple and explicit -variable functions that require at least lines to compute, see [this paper of Iwama et al](http://www.wisdom.weizmann.ac.il/~ranraz/publications/P5nlb.pdf) as well as this more recent [work of Kulikov et al](http://logic.pdmi.ras.ru/~kulikov/papers/2012_5n_lower_bound_cie.pdf). Proving lower bounds for restricted models of circuits is an extremely interesting research area, for which Jukna’s book [@Jukna12] (see also Wegener [@wegener1987complexity]) provides a very good introduction and overview. I learned of the proof of the size hierarchy theorem (sizehiearchythm) from Sasha Golovnev.

Scott Aaronson’s blog post on how [information is physical](http://www.scottaaronson.com/blog/?p=3327) is a good discussion on issues related to the physical extended Church-Turing Physics. Aaronson’s survey on NP complete problems and physical reality [@aaronson2005physicalreality] discusses these issues as well, though it might be easier to read after we reach cooklevinchap on and -completeness.

1. The implicit constant in the notation is smaller than . That is, for all sufficiently large , , see efficientrepresentation. As discussed in notationsec, we use the bound simply because it is a round number. [↑](#footnote-ref-35)
2. “Astronomical” here is an understatement: there are much fewer than stars, or even particles, in the observable universe. [↑](#footnote-ref-36)
3. The constant is at least and in fact, can be improved to be arbitrarily close to , see efficientlbex. [↑](#footnote-ref-38)
4. If you’re curious what these few lines are, see our [GitHub repository](https://github.com/boazbk/tcscode). [↑](#footnote-ref-52)
5. Python does not distinguish between lists and arrays, but allows constant time random access to an indexed elements to both of them. One could argue that if we allowed programs of truly unbounded length (e.g., larger than ) then the price would not be constant but logarithmic in the length of the array/lists, but the difference between and will not be important for our discussions. [↑](#footnote-ref-67)
6. ARM stands for “Advanced RISC Machine” where RISC in turn stands for “Reduced instruction set computer”. [↑](#footnote-ref-75)
7. We were extremely conservative in the suggested parameters for the PECTT, having assumed that as many as bits could potentially be stored in a millimeter radius region. [↑](#footnote-ref-96)
8. This is a very rough approximation that could be wrong to a few orders of magnitude in either direction. For one, there are other structures in the brain apart from neurons that one might need to simulate, hence requiring higher overhead. On the other hand, it is by no mean clear that we need to fully clone the brain in order to achieve the same computational tasks that it does. [↑](#footnote-ref-99)
9. There are some well known scientists that have [advocated](http://www.telegraph.co.uk/science/2017/03/14/can-solve-chess-problem-holds-key-human-consciousness/) that humans have inherent computational advantages over computers. See also [this](https://arxiv.org/abs/1508.05929). [↑](#footnote-ref-101)
10. How many functions from to exist? Note that our definition of circuits requires each output to correspond to a unique gate, though that restriction can make at most an additive difference in the number of gates. [↑](#footnote-ref-113)
11. Follow the proof of sizehiearchythm, replacing the use of the counting argument with countingmultibitex. [↑](#footnote-ref-115)
12. Using the adjacency list representation, a graph with in-degree zero vertices and in-degree two vertices can be represented using roughly bits. The labeling of the input and output vertices can be specified by a list of labels in and labels in . [↑](#footnote-ref-117)
13. *Hint:* Use the results of efficientrepresentationex and the fact that in this regime and . [↑](#footnote-ref-119)
14. **Hint:** An equivalent way to say this is that you need to prove that the set of functions that can be computed using at most lines has fewer than elements. Can you see why? [↑](#footnote-ref-122)
15. Note that if is big enough, then it is easy to represent such a pair using bits, since we can represent the program using bits, and we can always pad our representation to have exactly length. [↑](#footnote-ref-123)
16. *Hint:* Use our bound on the number of programs/circuits of size (program-count), as well as the Chernoff Bound ( chernoffthm) and the union bound. [↑](#footnote-ref-125)