Theory of Computation and Complexity

Muchang Bahng

Spring 2025

Contents

1 Prerequisites

We will introduce some common notations (which may or may not be consistent with mathematical notations) that are conventional in other texts.

Definition 1.1 ()

Given a set A, the set A^* is defined

$$A^* \equiv \bigcup_{i=1}^{\infty} \left(\prod_i A \right)$$

With this, the set of all binary numbers is $\{0,1\}^*$.

Definition 1.2 ()

The logarithm without any base will denote logarithm in base 2. That is,

$$\log n = \log_2 n$$

Definition 1.3 ()

A **decision problem** is a problem that can be posed as a yes-no question on an infinite set of inputs. A method for solving a decision problem, given in the form of an *algorithm*, is called a **decision procedure** for that problem. A decision problem which can be solved by an algorithm is called **decidable**.

It is traditional to define the decision problem as the set of possible inputs together with the set of inputs for which the answer is yes, and the set of inputs (i.e. the domain) can be numbers, floats, strings, etc.

Example 1.1 ()

Two examples of division problems are:

1. Deciding whether a given natural number is prime.
2. Given two numbers x and y, does x evenly divide y? The decision procedure can be long division.

The Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value of infinity. That is, if the time it takes for an algorithm to complete a problem with input size n is given by $f(n)$, then we say that the computational complexity is of the *order* $O(f(n))$. More formally, we can define it as such:

Definition 1.4 (Big-O Notation)

Let f and g be (nonnegative) real-valued functions both defined on the positive integers, and let $g(x)$ be strictly positive for all large enough values of x. One writes

$$f(x) = O\big(g(x)\big) \text{ as } x \to \infty$$

if the absolute value of $f(x)$ is at most a positive constant multiple of $g(x)$ for all sufficiently large values of x. That is, $f(x) = O\big(g(x)\big)$ if there exist positive integers M and n_0 such that

$$f(n) \leq Mg(n) \text{ for all } n \geq n_0$$

In many contexts, the assumption that we are interested in the growth rate as the variable x goes to

infinity is left unstated, and one write more simply that

$$f(x) = O\big(g(x)\big)$$

The O notation asymptotical; that is, it refers to very large x. This means that the contribution of the terms that grow "most quickly" will eventually make the other ones irrelevant, and so the following simplification rules can be simplified:

1. If $f(x)$ is a sum of several terms, if there is one with largest growth rate, it can be kept, and all others omitted.
2. If $f(x)$ is a product of several factors, any constants (terms in the product that do not depend on x) can be omitted.

Example 1.2 ()

Let there be a program that given input with length x, takes $f(x) = 6x^4 - 2x^3 + 5$ steps to solve whatever problem needs to be solved. Then, using the simplification steps above, we have

$$f(x) = O(x^4)$$

2 AON-CIRC and Straight Line Programs

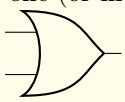
Definition 2.1 ()

The most common elementary operations for algorithms are **logical operators** which can be visualized as a **gate** in a **Boolean circuit**:

1. OR: $\{0,1\}^2 \longrightarrow \{0,1\}$, defined

$$OR(a,b) = a \vee b = \begin{cases} 0 & a = b = 0 \\ 1 & else \end{cases}$$

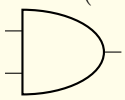
An **OR gate** has two incoming wires and one (or more) outgoing wires.



2. AND: $\{0,1\}^2 \longrightarrow \{0,1\}$, defined

$$AND(a,b) = a \wedge b = \begin{cases} 1 & a = b = 1 \\ 0 & else \end{cases}$$

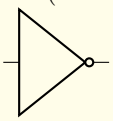
An **AND gate** has two incoming wires and one (or more) outgoing wires.



3. NOT: $\{0,1\} \longrightarrow \{0,1\}$, defined

$$NOT(a) = \neg a = \begin{cases} 0 & a = 1 \\ 1 & a = 0 \end{cases}$$

A **NOT gate** has one incoming wire and one (or more) outgoing wires)



Many functions can be created when composing these extremely simple functions.

Example 2.1 ()

Consider the function $MAJ : \{0,1\}^3 \longrightarrow \{0,1\}$ defined as follows

$$MAJ(x) = \begin{cases} 1 & x_0 + x_1 + x_2 \geq 2 \\ 0 & else \end{cases}$$

We can interpret this function as the following: $MAJ(x) = 1$ if and only if there exists some pair of distinct elements i, j such that both x_i and x_j are equal to 1. In other words, it means that $MAJ(x) = 1$ iff *either* both $x_0 = 1$ and $x_1 = 1$, *or* both $x_1 = 1$ and $x_2 = 1$, *or* both $x_0 = 1$ and $x_2 = 1$. Since the OR of three conditions c_0, c_1, c_2 can be written as

$$OR(c_0, OR(c_1, c_2))$$

we can now translate this function into a formula as follows:

$$\begin{aligned} MAJ(x_0, x_1, x_2) &= OR\big(AND(x_0, x_1), OR(AND(x_1, x_2), AND(x_0, x_2))\big) \\ &= \big((x_0 \wedge x_1) \vee (x_1 \wedge x_2)\big) \vee (x_0 \wedge x_2) \end{aligned}$$

Definition 2.2 ()

A **straight-line program** is a program that defines certain functions $F, G, H...$ and uses these programs to define variables of the form

```
foo = F(bar,blah)
foo = G(bar,blah)
foo = H(bar)
...  = ...
```

to come to a result. It is called a straight-line program since it contains no loops or branching (e.g. if/then statements).
The **AON-CIRC programming language** has the AND/OR/NOT operations defined. The binary input variables are of the form

$$x = (\texttt{X[0], X[1], ..., X[n-1]})$$

and output variables of the form

$$y = (\texttt{Y[0], Y[1], ..., Y[m-1]})$$

In every line, the variables on the right-hand side of the assignment operators must either be input variables or variables that have already been assigned a value. We say that an AON-CIRC program P *computes* a function

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

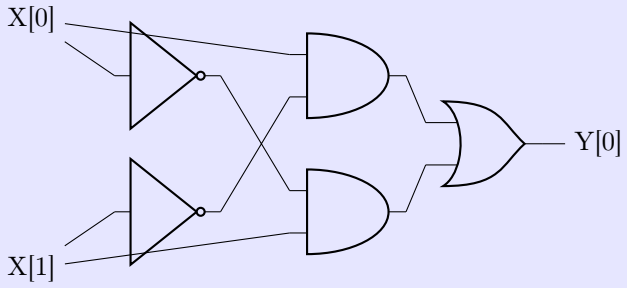
if $P(x) = f(x)$ for every $x \in \{0,1\}^n$.

Example 2.2 ()

Let the XOR function be defined

$$XOR : \{0,1\}^2 \longrightarrow \{0,1\},\ XOR(a,b) = a + b \pmod 2$$

The Boolean circuit for computing $XOR : \{0,1\}^2 \longrightarrow \{0,1\}$ is:



This can be computed with the straight-line algorithm as such. Given (a, b) as inputs, we have $w_1 = AND(a,b), w_2 = NOT(w_1)$, and $w_3 = OR(a,b)$. Then the algorithm returns $AND(w_2, w_3)$. In Python, this can be programmed:

```
def AND(a, b): return a*b
def OR(a, b): return 1-(1-a)*(1-b)
def NOT(a): return 1-a

def XOR(a, b):
    w1 = AND(a, b)
```

```
    w2 = NOT(w1)
    w3 = OR(a,b)
    return AND(w2, w3)

print([f"XOR({a},{b})={XOR(a,b)}" for a in [0,1] for b in [0,1]])
# ['XOR(0,0)=0', 'XOR(0,1)=1', 'XOR(1,0)=1', 'XOR(1,1)=0']
```

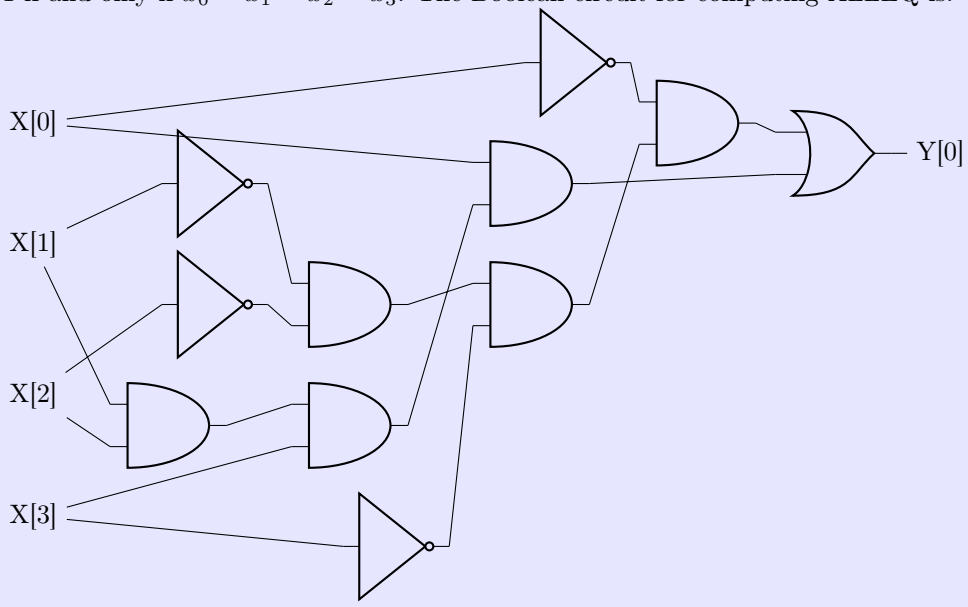
Note that Boolean circuits are a *mathematical model* that does not necessarily correspond to a physical object, but they can be implemented physically. In physical implementation of circuits, the signal is often implemented by electric potential, or voltage, on a wire, where for example voltage above a certain level is interpreted as a logical value of 1, and below a certain level is interpreted as a logical value of 0. Furthermore, the way that we've presented the XOR function through Boolean circuits and straight-line programs hints at the following:

Theorem 2.1 (Equivalence of circuits and straight line programs)

Let $f : \{0,1\}^n \longrightarrow \{0,1\}^m$ and $s \geq m$ be some number. Then f is computable by a Boolean circuit of s gates if and only if f is computable by an AON-CIRC program of s lines.

Example 2.3 ()

Let us define the function $ALLEQ : \{0,1\}^4 \longrightarrow \{0,1\}$ to be the function that on input $x \in \{0,1\}^4$ outputs 1 if and only if $x_0 = x_1 = x_2 = x_3$. The Boolean circuit for computing ALLEQ is:



2.1 Topological Sortings of Graphs

We now proceed to formally define Boolean circuits. But first, we must cover a few prerequisite definitions:

Definition 2.3 (Directed Graphs)

A **directed graph** $G = (V, E)$ consists of a set V and a set $E \subseteq V \times V$ of ordered pairs of V, which denotes the edge (u, v) or also as $u \to v$. If the edge $u \to v$ is present in the graph, then v is called

an **out-neighbor** of u and u is an **in-neighbor** of v.
The **in-degree** of u is the number of in-neighbors it has, and the **out-degree** of v is the number of out-neighbors it has. A **path** in the graph is a tuple $(u_0, u_1, ..., u_k) \in V^{k+1}$ for some $k > 0$ such that u_{i+1} is an out-neighbor of u_i for every $i \in [k]$. A *simple path* is a path $(u_0, ..., u_k)$ where all the u_i's are distinct, and a *cycle* is a path where $u_0 = u_k$.

Definition 2.4 (Directed Acyclic Graphs)

We say that $G = (V, E)$ is a **directed acyclic graph (DAG)** if it is a directed graph and there does not exist a list of vertices $u_0, u_1, ...u_k \in V$ such that $u_0 = u_k$ and for every $i \in [k]$, the edge $u_i \to u_{i+1}$ is in E.

Every directed acyclic graph can be arranged in layers so that for all directed edges $u \to v$, the layer of v is larger than the layer of u. This is expressed more formally in the following definition.

Definition 2.5 (Layering of a DAG)

Let $G = (V, E)$ be a directed graph. A **layering** of G is a function

$$f : V \longrightarrow \mathbb{N}$$

such that for every edge $u \to v$, $f(u) < f(v)$.

The next lemma is extremely useful.

Theorem 2.2 ()

Let G be directed graph. Then G is acyclic if and only if there exists a layering f of G. This is result is known as **topological sorting**.

Corollary 2.1 ()

There exists a layering for every directed acyclic graph. That is, every DAG can be topologically sorted.

2.1.1 Formal Definition of Boolean Circuits

Definition 2.6 ()

Let n, m, s be positive integers with $s \geq m$. A **Boolean circuit** with n inputs, m outputs, and s gates, is a labeled *directed acyclic graph* (DAG)

$$G = (V, E)$$

with $s + n$ vertices satisfying the following properties:

1. Exactly n of the vertices have no in-neighbors (i.e. inputs). These vertices known known as **inputs** and are labeled with the n labels

   $$X[0], X[1], ..., X[n-1]$$

   Each input has at least one out-neighbor.
2. The other s vertices are known as **gates**. Each gate is labeled with $\wedge, \vee$, or $\neg$. Gates labeled with $\wedge$ (AND) or $\vee$ (OR) have two in-neighbors. Gates labeled with $\neg$ (NOT) have one in-neighbor. **Parallel edges** are allowed.

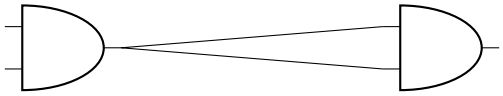
3. Exactly m of the gates are also labeled with the m labels

$$Y[0], Y[1], ..., Y[m-1]$$

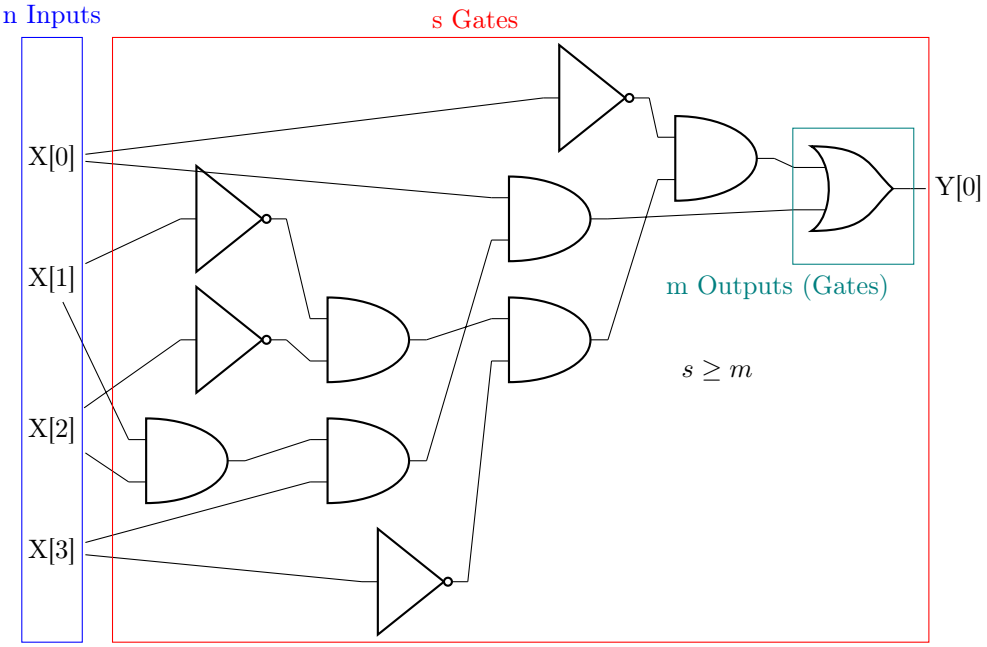
in addition to their label $\wedge/\vee/\neg$. These are known as **outputs**.

The **size** of a Boolean circuit is the number of gates it contains.

Having parallel edges means that an AND or OR gate u can have both its in-neighbors be the same gate v. Since $AND(a,a) = OR(a,a) = a$ for every $a \in \{0,1\}$, such parallel gates don't help in computing new values in circuits with AND/OR/NOT gates.



We clarify the definition with the previous example of the function ALLEQ.



Definition 2.7 ()

We can also see that a Boolean circuit naturally induces a function defined in the space $\{0,1\}^n$. That is, given Boolean circuit C with n inputs and m outputs, let the *output* of C on the input $x \in \{0,1\}^n$ be denoted $C(x)$. Then, if a function

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

satisfies $f(x) = C(x)$ for all $x \in \{0,1\}^n$, we say that the circuit C **computes** f.

2.2 Physical Implementations of Computing Devices

Note that *computation* is an abstract notion (a process) that is distinct from its physical *implementations* (how the progress is run). While most modern computing devices are obtained by mapping logical gates to semiconductor-based transistors, throughout history people have computed using a huge variety of mechanisms, including mechanical systems, gas and liquid (known as fluidics), biological and chemical processes,

and even living creatures. We will explore some ways that allow us to directly translate Boolean circuits to the physical world, without going through the entire stack of architecture, operating systems, and compilers.

2.2.1 Transistors

A transistor can be thought of as an electric circuit with two inputs, known as the source and the gate and an output, known as the sink. The gate controls whether current flows from the source to the sink. In a standard transistor, if the gate is "ON" then current can flow from the source to the sink and if it is "OFF" then it can't. In a complementary transistor this is reversed: if the gate is "OFF" then current can flow from the source to the sink and if it is "ON" then it can't.

We can use transistors to implement various Boolean functions such as and AND, OR, and NOT. For each a two-input gate $G : \{0,1\}^2 \longrightarrow \{0,1\}$, such an implementation would be a system with two input wires x, y and one output wire z, such that if we identify high voltage with 1 and low voltage with 0, then the wire z will equal to 1 if and only if applying G to the values of the wires x and y is 1.

Biological computing Computation can be based on biological or chemical systems. For example the lac operon produces the enzymes needed to digest lactose only if the conditions $x \wedge (\neg y)$ hold, where x is "lactose is present" and y is "glucose is present."

Cellular Automata and the Game of Life Cellular automata is a model of a system composed of a sequence of cells, each of which can have a finite state. At each step, a cell updates its state based on the states of its neighboring cells and some simple rules. As we will discuss later in this book, cellular automata such as Conway's *Game of Life* can be used to simulate computation gates .

Neural Networks Another computation device is the brain. Even though the exact working of the brain is still not fully understood, one common mathematical model for it is a (very large) **neural network**.

A neural network can be thought of as a Boolean circuit that instead of AND/OR/NOT uses some other gates as the basic basis. One particular basis we can use are **threshold gates**. For every vector

$$w = (w_0, w_1, ..., w_{k-1})$$

of integers and integer t (some or all of which could be negative), the **threshold function corresponding to** w, t is the function $T_{w,t} : \{0,1\}^k \longrightarrow \{0,1\}$ that maps $x \in \{0,1\}^k$ to 1 if and only if

$$\sum_{i=0}^{k-1} w_i x_i \geq t$$

that make up the core of human and animal brains. To a first approximation, a neuron has k inputs and a single output, and the neurons "fires" or "turns on" its output when those signals pass some threshold.

2.3 The NAND Function

Definition 2.8 ()

The NAND function is a function mapping $\{0,1\}^2$ to $\{0,1\}$ defined by

$$NAND(a,b) = \begin{cases} 0 & a = b = 1 \\ 1 & else \end{cases}$$

NAND is really the composition of the NOT and AND functions; that is,

$$NAND(a,b) = (NOT \circ AND)(a,b)$$

Here is an interesting result.

Theorem 2.3 (Universality of NAND)

We can compute AND, OR, and NOT by composing only the NAND function.

Proof.

We can see that, using double negation,

$$\begin{aligned} NOT(a) &= NOT(AND(a,a)) \\ &= NAND(a,a) \\ AND(a,b) &= NOT(NOT(AND(a,b))) \\ &= NOT(NAND(a,b)) \\ &= NAND(NAND(a,b), NAND(a,b)) \\ OR(a,b) &= NOT(AND(NOT(a), NOT(b))) \\ &= NOT(AND(NAND(a,a), NAND(b,b))) \\ &= NAND(NAND(a,a), NAND(b,b)) \end{aligned}$$

Corollary 2.2 ()

For every Boolean circuit C of s gates, there exists a NAND circuit C' of at most $3s$ gates that computes the same function as C.

Proof.

Replace every AND, OR, and NOT gate with their NAND equivalents.

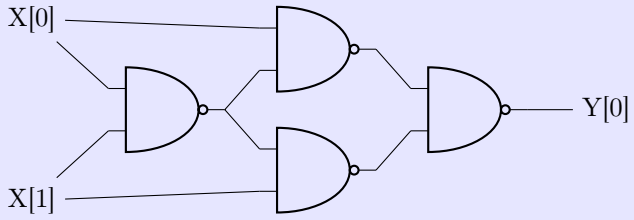
2.3.1 NAND Circuits

Definition 2.9 ()

A **NAND Circuit** is a circuit in which all the gates are NAND operations. Despite their simplicity, NAND circuits can be quite powerful.

Example 2.4 ()

We can create a NAND circuit of the XOR function that maps $x_0, x_1 \in \{0,1\}$ to $x_0 + x_1 \pmod 2$.



Definition 2.10 ()

Two models are said to be *equivalent in power* if they can be used to compute the same set of functions.

Just as we have defined the AON-CIRC program, we can define the notion of computation by a NAND-CIRC

program in the natural way.

Theorem 2.4 (Equivalence between models of finite computation)

For every sufficiently large s, n, m and $f : \{0,1\}^n \longrightarrow \{0,1\}^m$, the following conditions are all equivalent to one another:

1. f can be computed by a Boolean circuit (with $\wedge, \vee, \neg$ gates) of at most $O(s)$ gates.
2. f can be computed by an AON-CIRC straight-line program of at most $O(s)$ lines
3. f can be computed by a NAND circuit of at most $O(s)$ gates.
4. f can be computed by a NAND-CIRC straight-line program of at most $O(s)$ lines.

By $O(s)$, we mean that the bound is at most $c \cdot s$, where c is a constant that is independent of n. For example, if f can be computed by a Boolean circuit of s gates, then it can be computed by a NAND-CIRC program of at most $3s$ lines, and if f can be computed by a NAND circuit of s gates, then it can be computed by an AON-CIRC program of at most $2s$ lines.

2.3.2 Circuits with other Gate Sets

We can expand beyond the basis functions of AND/OR/NOT or NAND to a general set of functions

$$\mathcal{G} = \{G_0, G_1, ..., G_{k-1}\}$$

With this, we can define a notion of circuits that use elements of $\mathcal{G}$ as gates and a notion of a $\mathcal{G}$ *programming language* where every line involves assigning to a variable `foo` the result of applying some $G_i \in \mathcal{G}$ to previously defined or input variables. We state this formally.

Definition 2.11 (General Straight-line programs)

Let $\mathcal{F} = \{f_0, f_1, ..., f_{t-1}\}$ be a finite collection of Boolean functions such that

$$f_i : \{0,1\}^k \longrightarrow \{0,1\}$$

for some $k_i \in \mathbb{N}$. A $\mathcal{F}$ **program** is a sequence of lines, each of which assigns to some variable the result of applying some $f_i \in \mathcal{F}$ to k_i other variables. As above, we use `X[i]` and `Y[j]` to denote the input and output variables.

We say that $\mathcal{F}$ is a **universal set of operations** (or a **universal gate set**) if there exists a $\mathcal{F}$ program to compute the function NAND.

Example 2.5 ()

Let $\mathcal{F} = \{IF, ZERO, ONE\}$ where

$$ZERO : \{0,1\} \longrightarrow \{0\}, \quad ONE : \{0,1\} \longrightarrow \{1\}$$

are the constant zero and one functions, and

$$IF : \{0,1\}^3 \longrightarrow \{0,1\}, \; IF(a,b,c) = \begin{cases} b & a = 1 \\ c & else \end{cases}$$

Then, $\mathcal{F}$ is universal since we can use the following formula to compute NAND:

$$NAND(a,b) = IF\big(a, IF(b, ZERO, ONE), ONE\big)$$

There are some sets $\mathcal{F}$ that are more restricted in power. For example, it can be shown that if we use only AND or OR gates (without NOT), then we do not get an equivalent model of computation.

2.4 Syntactic Sugar

Just as we have built the AND, OR, and NOT gates with the NAND gate, we can implement more complex features using our basic building blocks, and then use these new features themselves as building blocks for even more sophisticated features. This is known as **syntactic sugar**, since we are not modifying the underlying programming model itself, but rather we merely implement new features by syntactically transforming a program that uses such features into one that doesn't. It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.

In computer programming, we can define and then execute **procedures** or **subroutines**, which are often known as *functions*.

Example 2.6 ()

We can use syntactic sugar to compute the majority function MAJ as follows, by first defining the procedures NOT, AND, and OR.

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

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

Note that compared to writing out the full Boolean circuit without any syntactic sugar, one with sugar will can be much simpler. It's the difference between having access to only NAND, or all of NAND, AND, OR, NOT.

Definition 2.12 ()

We call these the programming language NAND-CIRC augmented with the syntax above (for defining procedures) a **NAND-CIRC-PROC** program. Note that NAND-CIRC-PROC only allows *non-recursive* procedures (that is, procedures that take in its return value as its argument).

Since the procedures are defined using the NAND operator, it is trivial that for every NAND-CIRC-PROC program P, there exists a "sugar-free" NAND-CIRC program P' that computes the same function as P.

2.4.1 Conditional Statements

We can define conditional (if/then) statements using NAND operators. The idea is to compute the function $IF : \{0,1\}^3 \longrightarrow \{0,1\}$ such that $IF(a,b,c)$ equals b if $a = 1$ and c if $a = 0$.

Definition 2.13 ()

The IF function can be implemented from NANDs as follows:

```
def IF(cond, a, b);
```

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

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

With this, we can replace code of the form

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

with code of the form

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

that assigns to `foo` its old value when `condition` equals 0, and assign to `foo` the value of `blah` otherwise.

Definition 2.14 ()

Let NAND-CIRC-IF be the programming language NAND-CIRC augmented with `if/then/else` statements for allowing code to be conditionally executed based on whether a variable is equal to 0 or 1.

Theorem 2.5 ()

For every NAND-CIRC-IF program P, there exists a standard (i.e. "sugar-free") NAND- CIRC program P' that computes the same function as P.

2.4.2 Addition and Multiplication

We can write the integer addition function as follows:

```
# Add two n-bit integers
# Use LSB first notation for simplicity
def ADD(A,B):
    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 zero function, and `MAJ, XOR` correspond to the majority and XOR functions respectively. Note that in here, n is a *fixed integer* and so for every such n, `ADD` is a *finite* function that takes as input $2n$ bits and outputs $n + 1$ bits. Note that the `for` loop isn't anything fancy at all; it is just shorthand notation of simply repeating the code n times. By expanding out all the features, for every value of n we can translate the above program into a standard ("sugar-free") NAND-CIRC program. Note that the sugar free NAND-CIRC program to adding two-digit binary numbers consists of 43 lines of code, with a Boolean circuit of 15 layers.

We can in fact prove the following theorem that gives an upper bound on the addition algorithm.

Theorem 2.6 (Addition using NAND-CIRC programs)

For every $n \in \mathbb{N}$, let

$$ADD_n : \{0,1\}^{2n} \longrightarrow \{0,1\}^{n+1}$$

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, for every n there is a NAND-CIRC program to compute ADD_n with at most $9n$ lines.

Once we have addition, we can use grade-school algorith of multiplication to obtain multiplication as well.

Theorem 2.7 (Muliplication using NAND-CIRC programs)

For every n, let

$$MULT_n : \{0,1\}^{2n} \longrightarrow \{0,1\}^{2n}$$

be the function that, given $x, x' \in \{0,1\}^n$, computes the representation of the product of the numbers that x and x' represent. Then, there is a constant c such that for every n, there is a NAND-CIRC program of at most cn^2 that computes the function $MULT_n$.

The *Karatsuba's algorithm* allows us to actually compute that there is a NAND-CIRC program of $O(n^{\log_2 3})$ lines to compute $MULT_n$.

2.4.3 The Lookup Function

The LOOKUP function tells us the value of a certain entry.

Definition 2.15 (Lookup function)

For every k, the **lookup function** of order k,

$$LOOKUP_k : \{0,1\}^{2^k} \times \{0,1\}^k \simeq \{0,1\}^{2^k+k} \longrightarrow \{0,1\}$$

(where $\simeq$ denotes isomorphism) is defined as follows: For every $x \in \{0,1\}^{2^k}$ and $i \in \{0,1\}^k$,

$$LOOKUP_k(x, i) = x_i$$

where x_i denotes the ith entry of x in binary representation.

Theorem 2.8 ()

For every $k > 0$, there is a NAND-CIRC program that computes the function $LOOKUP_k : \{0,1\}^{2^k+k} \longrightarrow \{0,1\}$. The number of lines in this program is at most $4 \cdot 2^k$. This also means that $LOOKUP_k$ can be computed by a Boolean circuit (with AND, OR, and NOT) gates of at most $8 \cdot 2^k$ gates.

2.4.4 Computing Every Function

In fact, we can compute *every* finite function with a large enough Boolean (or equivalently, NAND) circuit.

Theorem 2.9 (Universality of Finite Functions)

There exists some constant $c > 0$ such that for every $n, m > 0$ and function

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

there is a NAND-CIRC program/NAND circuit, with at most $c \cdot m2^n$ lines/gates that computes the function f.

Since the models of NAND circuits, NAND-CIRC programs, and AON-CIRC programs, and Boolean circuits are all equivalent to one another, we can restate the theorem as such.

This may not be so surprising actually. After all, a finite function $f : \{0,1\}^n \longrightarrow \{0,1\}^m$ can be represented by simply the list of its outputs for each one of the 2^n input values. So it makes sense that we could write a NAND-CIRC program of similar size to compute it.

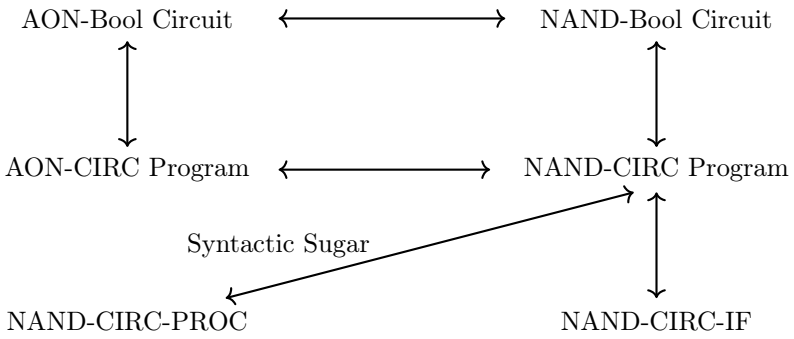
Definition 2.16 ()

For every $n, m \in \{1, 2, ..., 2s\}$, let $SIZE_{n,m}(s)$ denote the set of all functions $f : \{0,1\}^n \longrightarrow \{0,1\}^m$ such that $f \in SIZE(s)$. We denote $SIZE_n(s)$ to be just $SIZE_{n,1}(s)$. For every integer $s \geq 1$, we let

$$SIZE(s) = \bigcup_{n,m \leq 2s} SIZE_{n,m}(s)$$

be the set of all functions f that can be computed by NAND circuits of at most s gates (or equivalently, by NAND-CIRC programs of at most s lines).

We can summarize the equivalence of these models below:



3 Code as Data, Data as Code

A program is simply a sequence of symbols, each of which can be encoded in binary 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 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. That is, **a program is a piece of text, and so it can be fed as input to other programs**.

3.1 Representing Programs as Strings

We can represent programs or circuits as strings in many ways. For example, since Boolean circuits are labeled directed acyclic graphs, we can use the *adjacency matrix* representations. A simpler way is to just interpret the program as a sequence of letters and symbols. For example, the NAND-CIRC program P:

```
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, equality signs, punctuation marks, space, and the "new line" markers, all of which can be encoded in ASCII. Since every symbol can be encoded as a string of 7 bits using the ASCII encoding, the program P can be encoded as a string of length $7 \cdot 107 = 749$ bits. Therefore, we can prove that *every* NAND-CIRC program can be represented as a string in $\{0,1\}^*$.

Furthermore, 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 P', where all variables apart from the inputs and outputs, have the form $\texttt{temp}_0$, $\texttt{temp}_1$, ... Moreover, if the program has s, lines, then we will never need to use an index larger than $3s$ (since each line involves at most three variables), and similarly, the indices of the input and output variables will all be at most $3s$. Since a number between 0 and $3s$ can be expressed using at most $\lceil \log_{10}(3s+1) \rceil = O(\log s)$ digits, each line in the program (which has the form `foo = NAND(bar, blah)`), can be represented using $O(1) + O(\log s) = O(\log s)$ symbols, each of which can be represented by 7 bits. This results in the following theorem

Theorem 3.1 (Representing programs as strings)

There is a constant c such that for $f \in SIZE(s)$, there exists a program P computing f whose string representation has length at most $cs \log s$.

3.2 Counting Programs

We can actually see that the number of programs of certain length is bounded by the number of strings that represent them.

Theorem 3.2 (Counting programs)

For every $s \in \mathbb{N}$,

$$|SIZE(s)| \leq 2^{O(s \log s)}$$

That is, there are at most $2^{O(s \log s)}$ functions computed by NAND-CIRC programs of at most s lines. This gives a limitation on NAND-CIRC programs running on at most a given number of s lines.

Note that a function mapping $\{0,1\}^2 \longrightarrow \{0,1\}$ can be identified with a table of its four values on the inputs 00, 01, 10, 11. A function mapping $\{0,1\}^3 \longrightarrow \{0,1\}$ can be identified with the table of its 8 values on the inputs 000, 001, 010, 011, 100, 101, 110, 111. More generally, every function

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

is equal to the number of such tables which is 2^{2^n}. Note that this is a *double exponential* in n, and hence even form small values of n (e.g. $n = 10$), the number of functions from $\{0,1\}^n \longrightarrow \{0,1\}$ is large.

Theorem 3.3 (Counting argument lower bound)

The shortest NAND-CIRC program to compute $f : \{0,1\}^n \longrightarrow \{0,1\}$ requires more than $\delta \cdot 2^n/n$ lines. That is, there exists a constant $\delta > 0$ such that for every sufficiently large n, there exists $f : \{0,1\}^n \longrightarrow \{0,1\}$ such that $f \notin SIZE\left(\frac{\delta 2^n}{n}\right)$. The constant δ can be proven to be arbitrarily close to $\frac{1}{2}$.

We already know that every function mapping $\{0,1\}^n$ to $\{0,1\}$ can be computed by an $O(2^n/n)$ line program. The previous theorem shows that some functions do require an astronomical number of lines to compute. That is, **some functions $f : \{0,1\}^n \longrightarrow \{0,1\}$ cannot be computed by a Boolean circuit using fewer than exponential (in n) number of gates**.

3.3 Tuples Representation

ASCII is a fine representation of programs, but we can do better. That is, give a NAND-CIRC program with lines of the form

```
blah = NAND(baz, boo)
```

We can encode each line as the triple `(blah, baz, boo)`. Furthermore, we can associate each variable with a number and encode the line with the 3-tuple (i, j, k). Expanding on this, we can associate every variable with a number in the set

$$[t] = \{0, 1, 2, ..., t-1\}$$

where the first n numbers $\{0, ..., n-1\}$ correspond to input variable, the last m numbers $\{t-m, ..., t-1\}$ correspond to the output variables, and the intermediate numbers $\{n, ..., t-m-1\}$ correspond to the remaining variables.

Definition 3.1 (List of tuples representation)

Let P be a NAND-CIRC program of n inputs, m outputs, and s lines, and let t be the number of distinct variables used by P. The **list of tuples representation of** P is the triple (n, m, L), where L is the list of triples of the form (i, j, k) for $i, j, k \in [t]$. We assign a number for a variable of P as follows:

1. For every $i \in [n]$, the variable `X[i]` is assigned to the number i.
2. For every $j \in [m]$, the variable `Y[j]` is assigned to the number $t - m + j$.
3. Every other variable is assigned a number in $\{n, n+1, ..., t-m-1\}$ in the order in which the variable appears in the program P.

This is usually the default representation for NAND-CIRC programs, so we will call it "the representation" shorthand. The program could be represented as the list L instead of the triple (n, m, L).

Example 3.1 ()

To represent the XOR program of lines

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

we represent it as the tuple

$$L = \big((2,0,1), (3,0,2), (4,1,2), (5,3,4)\big)$$

Note that the variables `X[0], X[1]` are given the indices $0, 1$, the variable `Y[0]` is given the index 5, and the variables `u, v, w` are given the indices $2, 3, 4$.

So, if P is a program of size s, then the number t of variables is at most $3s$. Therefore, we can encode every variable index in $[t]$ as a string of length $l = \lceil \log(3s) \rceil$ (in binary), by adding leading zeros as needed. Since this is fixed-length encoding, it is prefix free, and so we can encode the list L of s triples as simply as the string of length $3ls$ obtained by concatenating all of these encodings.

Letting $S(s)$ be the length of the string representing the list L corresponding to a size s program, we get

$$S(s) = 3sl = 3s\lceil \log(3s) \rceil$$

3.4 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 $s, n, m > 0$, we define the function

$$EVAL_{s,n,m} : \{0,1\}^{S(s)+n} \longrightarrow \{0,1\}^m$$

as such: Given that px is the concatenation of two strings $p \in \{0,1\}^{S(s)}$ representing a list of triples L that represents a size-s NAND-CIRC program P, and $x \in \{0,1\}^n$ is a string,

$$EVAL_{s,n,m}(px) = P(x)$$

where $P(x)$ is equal to the evaluation $P(x)$ of the program P on input x. If p is not the list of tuples representation of a NAND-CIRC program, then $EVAL_{s,n,m} = 0^m$ (error message). Some important properties of EVAL include:

1. $EVAL_{s,n,m}$ is a finite function takin a string of fixed length as input and outputting a string of fixed length as output.
2. $EVAL_{s,n,m}$ is a single function, such that computing $EVAL_{s,n,m}$ allows us to evaluate *arbitrary* NAND-CIRC programs of a certain lenfth on *arbitrary* inputs of the appropriate length.
3. $EVAL_{s,n,m}$ is a *function*, not a *program*. That is, $EVAL_{s,n,m}$ is a *specification* of what output is associated with what input. The existence of a *program* that computes $EVAL_{s,n,m}$ (i.e. an *implementation* for $EVAL_{s,n,m}$) is a separate fact, which needs to be established.

Theorem 3.4 ()

For every $s, n, m \in \mathbb{N}$ with $s \geq m$, there is a NAND-CIRC program $U_{s,n,m}$ that computes the function $EVAL_{s,n,m}$.

That is, the NAND-CIRC program $U_{s,n,m}$ takes the description of *any other NAND-CIRC program* P (of the right length and inputs/outputs) and *any input* x, and computes the result of evaluating the program P on the input x. Given the equivalence between NAND-CIRC programs and Boolean circuits, we can also think of $U_{s,n,m}$ as a circuit that takes as inputs the description of other circuits and their inputs, and returns their evaluation.

Definition 3.2 ()

We call this NAND-CIRC program $U_{s,n,m}$ that computes $EVAL_{s,n,m}$ a **bounded universal program**, or a **universal circuit**. It is "universal" in the sense that this is a *single program* that can evaluate arbitrary code, where "bounded" stands for the fact that $U_{s,n,m}$ only evaluates programs of bounded size.

This theorem is profound because it proves the existence of a NAND-CIRC program that takes in *another* NAND-CIRC program along with its input. But it provides no explicit bound on the size of this program. The following theorem takes care of that.

Theorem 3.5 (Efficient bounded universality of NAND-CIRC programs)

For every $s, n, m \in \mathbb{N}$, there is a NAND-CIRC program of at most $O(s^2 \log s)$ lines that computes the function

$$EVAL_{s,n,m} : \{0,1\}^{S+n} \longrightarrow \{0,1\}^m$$

defined above (where S is the number of bits needed to represent programs of s lines). This allows us to place an upper bound on the size of $U_{s,n,m}$ that is *polynomial* in its input length.

4 Infinite Functions, Automata, Regular Expressions

We now extend our definition of computational tasks to consider functions with the *unbounded* domain of $\{0,1\}^*$. Note that an infinite function F does not necessarily take input strings of infinite length, but rather ones that can be arbitrarily long.

The big takeaway from this chapter is that we can think of an algorithm as a "finite answer to an infinite number of questions." To express an algorithm, we need to write down a finite set of instructions that will enable us to compute on arbitrarily long inputs.

4.1 Functions with Inputs of Unbounded Length

Example 4.1 ()

Note that the function $XOR : \{0,1\}^* \longrightarrow \{0,1\}$ equals 1 iff the number of 1's in x is odd. At best, we can compute XOR_n, the restriction of XOR to $\{0,1\}^n$ with NAND-CIRC programs.

Example 4.2 ()

The multiplication function takes the binary representation of a pair of integers $x, y \in \mathbb{N}$ and outputs the binary representation of the product $x \cdot y$.

$$MULT : \{0,1\}^* \times \{0,1\}^* \longrightarrow \{0,1\}^*$$

Since we can represent a pair of strings as a single string, we will consider functions such as MULT as

$$MULT : \{0,1\}^* \longrightarrow \{0,1\}^*$$

Example 4.3 (Palindrome function)

Another example of an infinite function is

$$PALINDROME(x) = \begin{cases} 1 & \forall i \in ||x||,\ x_i = x_{|x|-i} \\ 0 & else \end{cases}$$

which outputs 1 if x is a (base-2) palindrome and 0 if not.

Definition 4.1 ()

Sometimes, we can obtain a Boolean variant of a non-Boolean function. This process is called **booleanizing**.

Example 4.4 (Boolean variant of MULT)

The following is a boolean variant of MULT

$$BMULT(x, y, i) = \begin{cases} i\text{th bit of } x \cdot y & i < |x \cdot y| \\ 0 & else \end{cases}$$

Note that if we can compute $BMULT$, we can compute MULT as well, and vice versa.

4.2 Deterministic Finite Automata

Definition 4.2 ()

A **single-pass constant-memory algorithm** is an algorithm that computes an output from an input via a combination of the following steps:

1. Read a bit from the input.
2. Update the *state* (working memory).
3. Repeat the first 2 steps to pass over the input.
4. Stop and produce an output.

It is called "single-pass" since it makes a single pass over the input and "constant-memory" since its working memory is finite. Such an algorithm is also known as a **Deterministic Finite Automaton (DFA)**, or a **finite state machine**.

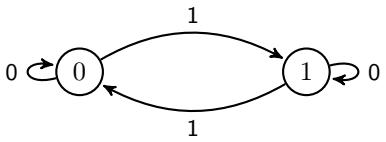
We can think of such an algorithm as a "machine" that can be in one of C states, for some constant C. The machine starts in some initial state and then reads its input $x \in \{0,1\}^*$ one bit at a time. Whenever the machine reads a bit $\sigma \in 0, 1$, it transitions into a new state based on σ and its prior state. The output of the machine is based on the final state. Every single-pass constant-memory algorithm corresponds to such a machine. If an algorithm uses c bits of memory, then the contents of its memory can be represented as a string of length c. Therefore such an algorithm can be in one of at most 2^c states at any point in the execution.

We can specify a DFA of C states by a list of $2C$ rules. Each rule will be of the form "If the DFA is in state v and the bit read from the input is σ then the new state is v'". At the end of the computation, we will also have a rule of the form "If the final state is one of the following ... then output 1, otherwise output 0".

For example, the Python program above can be represented by a two-state automaton for computing XOR of the following form:

1. Initialize in the state 0
2. For every state $s \in \{0,1\}$ and input bit σ read, if $\sigma = 1$, then change to state $1 - s$, otherwise stay in state s
3. At the end, output 1 iff $s = 1$

It can also be represented in the following graph.



More generally, a C-state DFA can be represented as a labeled graph of C nodes. The set $\mathcal{S}$ of states on which the automaton will output 1 at the end of the computation is known as the set of **accepting states**. We formally summarize it below.

Definition 4.3 ()

A **deterministic finite automaton (DFA)** with C states over $\{0,1\}$ is a pair $(T, \mathcal{S})$ with

$$T : [C] \times \{0,1\} \longrightarrow [C]$$

and $\mathcal{S} \subset [C]$. The finite function T is known as the **transition function** of the DFA. The set $\mathcal{S}$ is known as the set of **accepting states**.
Let $F : \{0,1\}^* \longrightarrow \{0,1\}$ be a Boolean function with the infinite domain $\{0,1\}^*$. We say that $(T, \mathcal{S})$ **computes** a function $F : \{0,1\}^* \longrightarrow \{0,1\}$ if for every $n \in \mathbb{N}$ and $x \in \{0,1\}^n$, if we define $s_0 = 0$ and $s_{i+1} = T(s_i, x_i)$ for every $i \in [n]$, then

$$s_n \in \mathcal{S} \iff F(x) = 1$$

Note that the transition function T is a finite function specifying the table of "rules" for which the graph evolves. By defining the DFA C with $(T, \mathcal{S})$, we have essentially reduced a specific type of infinite Boolean function (a single-pass constant-memory algorithm) into a graph and a finite transition function.

When constructing a deterministic finite automaton, it helps to start by thinking of it as a single-pass constant-memory algorithm, and then translate this program into a DFA.

Definition 4.4 ()

We say that a function $F : \{0,1\}^* \longrightarrow \{0,1\}$ is **DFA computable** if there exists some DFA that computes F.

Theorem 4.1 ()

Let $DFACOMP$ be the set of all Boolean functions $F : \{0,1\}^* \longrightarrow \{0,1\}$ such that there exists a DFA computing F. Then, $DFACOMP$ is countable.

Lemma 4.1 ()

The set of all Boolean functions $\{f \mid f : \mathbb{N} \longrightarrow \{0,1\}\}$ are uncountable.

Corollary 4.1 (Existence of DFA-uncomputable functions)

There exists a Boolean function $F : \{0,1\}^* \longrightarrow \{0,1\}$ that is not computable by *any* DFA.

4.3 Regular Expressions

Searching for a piece of text is a common task in computing. At its heart, the *search problem* is quite simple. We have a collection $X = \{x_0, ..., x_k\}$ of strings (e.g. files on a hard-drive, or student records in a database), and the user wants to find out the subset of all the $x \in X$ that are *matched* by some pattern. In full generality, we can allow the user to specify the pattern by specifying a (computable) function $F : \{0,1\}^* \longrightarrow \{0,1\}$, where $F(x) = 1$ corresponds to the pattern matching x. That is, the user provides a program P and the system returns all $x \in X$ such that $P(x) = 1$.

However, we don't want our system to get into an infinite loop just trying to evaluate the program P. For this reason, typical systems for searching files or databases do not allow users to specify the patterns using full-fledged programming languages. Rather, such systems use restricted computational models that on the one hand are rich enough to capture many of the queries needed in practice, but on the other hand are restricted enough so that queries can be evaluated very efficiently on huge files and in particular cannot result in an infinite loop. One of the most popular such computational models is *regular expressions*.

Definition 4.5 ()

A **regular expression** e over an alphabet Σ is a string over $\Sigma \cup \{(,), |, *, \emptyset, ""\}$ that has one of the following forms:

1. $e = \sigma$ where $\sigma \in \Sigma$
2. $e = (e' \,|\, e'')$ where e', e'' are regular expressions
3. $e = (e')(e'')$ where e', e'' are regular expressions. The parentheses are often dropped, so this is written $e'\, e''$
4. $e = (e')^*$ where e' is a regular expression

Finally, we also allow the following "edge cases": $e = \emptyset$ and $e = ""$. These are the regular expressions corresponding to accepting no strings and accepting only the empty string, respectively.

Example 4.5 ()

The following are regular expressions over the alphabet $\{0, 1\}$.

$$\big(00(0^*)|11(1^*)\big)^* \qquad 00^*|11$$

Every regular expression e corresponds to a function $\Phi_e : \Sigma^* \longrightarrow \{0, 1\}$ where $\Phi_e(x) = 1$ if x *matches* the regular expression. The definition is tedious.

Definition 4.6 ()

Let e be a regular expression over the alphabet Σ. The function $\Phi_e : \Sigma^* \longrightarrow \{0, 1\}$ is defined as follows:

1. If $e = \sigma$, then $\Phi_e(x) = 1$ iff $x = \sigma$
2. If $e = (e' \,|\, e'')$, then $\Phi_e(x) = \Phi_{e'}(x) \vee \Phi_{e''}(x)$ where $\vee$ is the OR operator.
3. If $e = (e')(e'')$, then $\Phi_e(x) = 1$ iff there is some $x', x'' \in \Sigma^*$ such that x is the concatenation of x' and x'' and $\Phi_{e'}(x') = \Phi^{e''}(x'') = 1$
4. If $e = (e')^*$ then $\Phi_e(x) = 1$ iff there is some $k \in \mathbb{N}$ and some $x_0, x_1, ..., x_{k-1} \in \Sigma^*$ such that x is the concatenation $x_0, x_1, ..., x_{k-1}$ and $\Phi_{e'}(x_i) = 1$ for every $i \in [k]$.
5. For the edge cases, $\Phi_\emptyset$ is the 0 function, and $\Phi_{""}$ is the function that only outputs 1 on the empty string "".

It is said that a regular expression e over Σ **matches** a string $x \in \Sigma^*$ if $\Phi_e(x) = 1$.

A Boolean function is called *regular* if it outputs 1 on precisely the set of strings that are matched by some regular expression.

Definition 4.7 ()

Let Σ be a finite set and $F : \Sigma^* \longrightarrow \{0, 1\}$ be a Boolean function. We say that F is **regular** if $F = \Phi_e$ for some regular expression e.
Similarly, for every formal language $L \subset \Sigma^*$, we say that L is regular if and only if there is a regular expression e such that $x \in L$ iff e matches x.

Definition 4.8 ()

The set of functions computable by DFAs is the same as the set of languages that can be recognized by regular expressions.

5 Turing Machines

Similar to how a person does calculations by reading from and writing to a single cell of a paper at a time, a Turing machine is a hypothetical machine that reads from its "work tape" a single symbol from a finite alphabet Σ and uses that to update its state, write to tape, and possibly move to an adjacent cell. To compute a function F using this machine, we initialize the tape with the input $x \in \{0,1\}^*$ and our goal is to ensure that the tape will contain the value $F(x)$ at the end of the computation. Specifically, a computation of a Turing machine M with k states and alphabet Σ on input $x \in \{0,1\}^*$ is formally defined as follows.

Definition 5.1 (Turing Machine)

A (one tape) **Turing machine** with k states and alphabet $\Sigma \supset \{0,1,\triangleright,\emptyset\}$ is represented by a **transition function**

$$\delta_M : [k] \times \Sigma \longrightarrow [k] \times \Sigma \times \{\mathsf{L},\mathsf{R},\mathsf{S},\mathsf{H}\}$$

For every $x \in \{0,1\}^*$, the *output* of M on input x, denoted by $M(x)$, is the result of the following process:

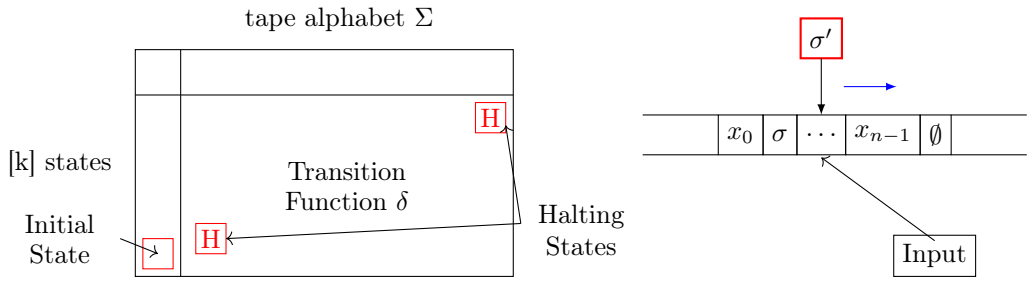
1. We initialize T to be the infinite sequence (also represented by a tape)

   $$\triangleright, x_0, x_1, ..., x_{n-1}, \emptyset, \emptyset, ...$$

   where $n = |x|$. That is, $T[0] = \triangleright, T[i+1] = x_i$ for $i \in [n]$, and $T[i] = \emptyset$ for $i > n$.)
2. We also initialize $i = 0$ (the head is at the starting position) and we begin with the initial state $s = 0$, $s \in [k]$.
3. We then repeat the following process which is defined according to the transition function:
   (a) Let $(s', \sigma', D) = \delta_M(s, T[i])$.
   (b) Set $s \to s'$, $T[i] \to \sigma'$
   (c) If $D = \mathsf{R}$, then set $i \to i+1$, if $D = \mathsf{L}$, then set $i \to \max\{i-1, 0\}$. If $D = \mathsf{S}$, then we keep i the same.
   (d) If $D = \mathsf{H}$, then halt.

   Colloquially, at each step, the machine reads the symbol $\sigma \in T[i]$ that is in the ith location of the tape. Bsaed on this symbol and its state s, the machine decides on
   (a) What symbol σ' to write on the tape
   (b) Whether to move Left ($i \to i-1$), Right ($i \to i+1$), Stay in place, or Halt the computation
   (c) What is going to be the new state $s \in [k]$
4. If the process above halts, then M's output, denoted by $M(x)$ is the string $y \in \{0,1\}^*$ obtained by concatenating all the symbols in $\{0,1\}$ in positions $T[0], ..., T[i]$ where $i+1$ is the first location in the tape containing $\emptyset$.
5. If the Turing machine does not halt then we denote $M(x) = \perp$.

We can visualize a Turing machine as a table and a tape labeled below.



In fact, all modern computing devices are Turing machines at heart. You input a string of bits, the machine flips a bunch of switches, and outputs another string of bits.

Example 5.1 (Turning Machine for Palindromes)

Let PAL be the function that on input $x \in \{0,1\}^*$, outputs 1 if and only if x is an (even length) *palindrome*, in the sense that

$$x = w_0 ... w_{n-1} w_{n-1} w_{n-2} ... w_0$$

for some $n \in \mathbb{N}$ and $w \in \{0,1\}^*$. We will now describe a Turing machine that computes PAL. To specify M, we need to specify

1. M's tape alphabet Σ which should contain at least the symbols 0, 1, $\triangleright$, and $\emptyset$, and
2. M's transition function which determines what action M takes when it reads a given symbol while it is in a particular state.

For this specific Turing machine, we will use the alphabet $\{0, 1, \triangleright, \emptyset, \times\}$ and will have $k = 13$ states, with the following labels for the numbers.

State	Label	State	Label
0	`START`	7	`ACCEPT`
1	`RIGHT_0`	8	`OUTPUT_0`
2	`RIGHT_1`	9	`OUTPUT_1`
3	`LOOK_FOR_0`	10	`0_AND_BLANK`
4	`LOOK_FOR_1`	11	`1_AND_BLANK`
5	`RETURN`	12	`BLANK_AND_STOP`
6	`REJECT`		

The operation of our Turning machine, in words, is as such:

1. M starts in the state `START` and goes right, looking for the first symbol that is 0 or 1. If it finds $\emptyset$ before it hits such a symbol then it moves to the `OUTPUT_1` state.
2. Once M finds such a symbol $b \in \{0,1\}$, M deletes b from the tape by writing the $\times$ symbol, it enters either the `RIGHT_0` or `RIGHT_1` mode according to the value of b and starts moving rightwards until it hits the first $\emptyset$ or $\times$ symbol.
3. Once M finds this symbol, it goes into the state `LOOK_FOR_0` or `LOOK_FOR_1` depending on whether it was in the state `RIGHT_0` or `RIGHT_1` and makes one left move.
4. In the state `LOOK_FOR_`b, M checks whether the value on the tape is b. If it is, then M deletes it by changing its value to $\times$, and moves to the state `RETURN`. Otherwise, it changes to the `OUTPUT_0` state.
5. The `RETURN` state means that M goes back to the beginning. Specifically, M moves leftward until it hits the first symbol that is not 0 or 1, in which case it changes its state to `START`.
6. The `OUTPUT_`b states mean that M will eventually output the value b. In both the `OUTPUT_0` and `OUTPUT_1` states, M goes left until it hits $\triangleright$. Once it doe sso, it makes a right step and changes to the `1_AND_BLANK` or `0_AND_BLANK` state respectively. In the latter states, M writes the corresponding value, moves right and changes to the `BLANK_AND_STOP` state, in which it writes $\emptyset$ to the tape and halts.

The above description can be turned into a table describing for each one of the $13 \cdot 5 = 65$ combinations of state and symbol, what the Turing machine will do when it is in that state and it reads that symbol. This table is the *transition function* of the Turing machine.

Definition 5.2 (Computable Functions)

Let $F : \{0,1\}^* \longrightarrow \{0,1\}^*$ be a (total) function and let M be a Turing machine. We say that M **computes** F if for every $x \in \{0,1\}^*$, $M(x) = F(x)$. We say that a function F is **computable** if there exists a Turing machines M that computes it.

It turns out that being computable in the sense of a Turing machine is equivalent to being computable in virtually any reasonable model of computation. This statement is known as the **Church-Turing Thesis**. Therefore, this definition allows us to precisely define what it means for a function to be computable by *any possible algorithm*.

Definition 5.3 (The class R)

We define **R** to be the set of all computable functions $F : \{0,1\}^* \longrightarrow \{0,1\}$.

5.1 NAND-TM Programs

In addition to having a physical interpretation, Turing machines can also be interpreted as programs.

1. The *tape* becomes a *list* or *array* that can hold values from the finite set Σ.
2. The *head position* can be thought of as an integer-valued variable that holds integers of unbounded size.
3. The *state* is a *local register* that can hold one of a fixed number of values in $[k]$.

In general, every Turing machine M is equivalent to a program similar to the following:

```
#Gets an array Tape initialized to [">", x_0,..., x_(n-1), " ", " ", ...]
def M(Tape):
    state = 0
    i     = 0  #holds head location
    while(True):
        #Move head, modify state, write to tape based on current state and
        #cell at head below are just examples for how program looks
        #for a particular transition function
        if Tape[i]=="0" and state==7: #T_M(7,"0")=(19,"1","R")
            i += 1
            Tape[i]="1"
            state = 19
        elif Tape[i]==">" and state == 13: #T_M(13,">")=(15,"0","S")
            Tape[i] ="0"
            state = 15
        elif...
            ...
        elif Tape[i]==">" and state == 29: #T_M(29,">")=(.,.,"H")
            break  #Halt
```

If we were using Boolean variables, then we can encode the `state` variables using $\lceil \log k \rceil$ bits.

Note that in the code above, two new concepts are introduced:

1. *Loops*: NAND-CIRC is a straight line programming language. That is, a NAND-CIRC program of s lines takes exactly s steps of computation and hence in particular, cannot even touch more than $3s$ variables. *Loops* allow us to use a fixed-length program to encode the instructions for a computation that can take an arbitrary amount of time.
2. *Arrays*: A NAND-CIRC program of s lines touches at most $3s$ variables. While we can use variables with names such as `Foo_17` or `Bar[22]` in NAND-CIRC, they are not true arrays, since the number in the identifier is a constant that is not "hardwired" into the program. NAND-TM contains actual arrays that can have a length that is not a priori bounded.

The following equation summarizes the concepts:

$$\text{NAND-TM} = \text{NAND-CIRC} + \text{loops} + \text{arrays}$$

Surprisingly, adding loops and arrays to NAND-CIRC is enough to capture the full power of all programming languages. Hence, we could replace NAND-TM with any of Python, C, Javascript, etc.

Concretely, the NAND-TM programming language adds the following features on top of NAND-CIRC:

1. We add a special *integer valued variable* i. All other variables in NAND-TM are Boolean valued (as in NAND-CIRC).
2. Apart from i, NAND-TM has two kinds of varibales: *scalars* and *arrays*. *Scalar* variables hold one bit (just as in NAND-CIRC). *Array* variables hold an unbounded number of bits. At any point in the computation we can access the array variables at the location indexed by `i` using `Foo[i]`. We cannot access the arrays at loctions other than the one pointed by `i`.
3. We use the convention that *arrays* always start with a capital letter, and *scalar variables* (which are never indexed with `i`) start with lowercase letters. Hence, `Foo` is an array and `foo` is a scalar variable.
4. The input and output `X` and `Y` are not considered *arrays* with values of 0s and 1s.
5. We add a special `MODANDJUMP` instruction that takes two Boolean variables a, b as input and does the following:
   (a) If $a = 1, b = 1$, then `MODANDJUMP(a, b)` increments `i` by one and jumps to the first line of the program.
   (b) If $a = 0, b = 1$, then `MODANDJUMP`(a, b) decrements `i` by one and jumps to the first line of the program. If `i` already equals 0, then it stays at 0.
   (c) If $a = 1, b = 0$, then `MODANDJUMP`(a, b) jumps to the first line of the program without modifying `i`.
   (d) If $a = b = 0$, then `MODANDJUMP`(a, b) halts execution of the program.
6. The `MODANDJUMP` instruction always appears in the last line of a NAND-TM program and nowhere else.
7. Turing machines have the special symbol $\emptyset$ to indicate that tape location is "blank" or "uninitialized." In NAND-TM there is no such symbol, and all variables are *Boolean*, containing either 0 or 1. All variables and locations either default to 0 if they have not been initialized to another value. To keep track of whether a 0 in an array corresponds to a true 0 or to an uninitialized cell, a programmer can always add to an array `Foo` a *companion array* `Foo_nonblank` and set `Foo_nonblank[i]` to 1 whenever the `i`th location is initialized. In particular, we will use this convention for the input and output arrays `X` and `Y`. Therefore, a NAND-TM program has *four* special arrays `X, X_nonblank, Y, Y_nonblank`.

Therefore, when a NAND-TM program is executed on input $x \in \{0,1\}^*$ of length n, the first n cells of `X` are initialized to $x_0, ..., x_{n-1}$ and the first n cells of `X_noblank` are initalized to 1 (all uninitialized cells default to 0). The output of a NAND-TM program is the string `Y[0], ..., Y[m-1]` where m is the smallest integer such that `Y_nonblank[m]` $= 0$.

We now formally define a NAND-TM program.

Definition 5.4 (NAND-TM Programs)

A **NAND-TM program** consists of a sequence of lines of the form `foo = NAND(bar, blah)` and ending with a line of the form `MODANDJMP(foo, bar)`, where `foo, bar, blah` are either *scalar variables* (sequence of letters, digits, and underscores) or *array variables* of the form `Foo[i]` (starting with capital letters and indexed by `i`). The program has the array variables `X, X_nonblank`, `Y, Y_nonblank` and the index variables `i` built in, and can use additional array and scalar variables. If P is a NAND-TM program and $x \in \{0,1\}^*$ is an input then an execution of P on x is the following process:

1. The arrays `X` and `X_nonblank` are initialized by `X[i]` $= x_i$ and `X_nonblank[i]` $= 1$ for all $i \in [|x|]$. All other variables and cells are initialized to 0. The index variable `i` is also initialized to 0.
2. The program is executed line by line. When the last line `MODANDJMP(foo, bar)` is executed we do as follows:
   (a) If `foo, bar` $= 1, 0$, jump to the first line without modifying the value of `i`.
   (b) If `foo, bar` $= 1, 1$, incremenet `i` by one and jump to the first line.

(c) If `foo, bar` $= 0, 1$, then decrement `i` by one (unless it is already 0) and jump to the first line.
(d) If `foo, bar` $= 0, 0$, halt and output `Y[0], ..., Y[m-1]` where m is the smallest integer such that `Y_nonblank[m]` $= 0$.

Here are some components of Turing machines and their analogs in NAND-TM programs.

1. The *state* of a Turing machine is equivalent to the *scalar-variables* such as `foo, bar, etc.`, each taking values in $\{0, 1\}$.
2. The *tape* of a Turing machines is equivalent to the *arrays*, where the component of each array is either 0 or 1.
3. The *head location* is equivalent to the *index variable*
4. *Accessing memory*: At every step the Turing machine has access to its local state, but can only access the tape at the position of the current head location. In a NAND-TM program, it has access to all the scalar variables, but can only access the arrays at the location `i` of the index variable.
5. A Turing machine can move the head location by at most one position in each step, while a NAND-TM program can modify the index `i` by at most one.

Theorem 5.1 (Equivalence of Turing Machines and NAND-TM programs)

For every function $F : \{0,1\}^* \longrightarrow \{0,1\}^*$, F is computable by a NAND-TM program P if and only if there is a Turing machine M that computes F.

Setting	Specification	Implentation
Finite Computation	$F : \{0,1\}^n \to \{0,1\}^m$	Circuit, Straightline program
Infinite Computation	$F : \{0,1\}^* \to \{0,1\}^*$	Algorithm, Turing Machine, Program

Finally, we can use syntactic sugar to make NAND-TM programs easier to write. For starters, we can use all of the syntactic sugar of NAND-CIRC, such as macro definitions and conditionals (if/then). However, we can go beyond this and achieve:

1. Inner loops such as the `while` and `for` operations common to many programming languages.
2. Multiple index variables (e.g. not just `i` but also `j, k`, etc.).
3. Arrays with more than one dimension (e.g., `Foo[i][j]`).

This means that the set of functions computable by NAND-TM with this feature is the same as the set of functions computable by standard NAND-TM.

5.1.1 Uniformity of Computation

Definition 5.5 ()

The notion of a single algorithm that can compute functions of all input length is known as **uniformity** of computation.

Hence we think of Turing machines and NAND-TM as *uniform* models of computation, as opposed to Boolean circuits of NAND-CIRC, which are non-uniform models, in which we have to specify a different program for every input length. This uniformity leads to another crucial difference between Turing machines and circuits. Turing machines can have inputs and outputs that are longer than the description of the machine as a string, and in particular there exists a Turing machine that can "self replicate" in the sense that it can print its own code. This is extremely useful.

In summary, the main differences between uniform and non-uniform models are described as such:

1. **Non-uniform computational models**: Examples are NAND-CIRC programs and Boolean circuits. These are models where each individual program/circuit can compute a *finite* function
$$f : \{0,1\}^n \longrightarrow \{0,1\}^m$$
We have seen that *every* finite function can be computed by *some* program/circuit. To discuss computation of an *infinite* function $F : \{0,1\}^* \longrightarrow \{0,1\}^*$, we need to allow a *sequence* $\{P_n\}_{n\in\mathbb{N}}$ of programs/circuits (one for every input length), but this does not capture the notion of a *single algorithm* to compute the function F.

2. **Uniform computational models**: Examples are Turing machines and NAND-TM programs. These are models where a single program/Turing machine can take inputs of *arbitrary length* and hence compute an *infinite* function
$$F : \{0,1\}^* \longrightarrow \{0,1\}^*$$
The number of steps that a program/machine takes on some input is not a priori bounded in advance and in particular there is a chance that it will enter into an *infinite loop*. Unlike the non-uniform case, we have *not* shown that every infinite function can be computed by some NAND-TM program/Turing machine.

5.2 RAM Machines and NAND-RAM Programs

Note that since Turing machines (and NAND-TM programs) can only access one locations of arrays/tape at a time, they do not have *RAM*.

Definition 5.6 ()

The computational model that models access to such a memory is the **RAM machine**. The **memory** of a RAM machine is an array of unbounded size where each cell can store a single **word**, which can be thought of as a string in $\{0,1\}^\omega$ and also (equivalently) as a number in $[2^\omega]$.

For example, many modern computing architectures use 64-bit words, in which every memory location holds a string in $\{0,1\}^{64}$. The parameter ω is known as the *word size*. In addition to the memory array, a RAM machine also contains a constant number of **registers** $r_0, r_1, ..., r_{k-1}$, each of which can also contain a word.

The oeprations a RAM machine can carry out include:

1. **Data movement**: Load data from a certain cell in memory into a register or store the contents of a register into a certain cell of memory. A RAM machine can directly access any cell of memory without having to move the "head" (as Turing machines do) to that location. That is, in one step a RAM machine can load into register r_i the contents of the memory cell indexed by register r_j, or store into the memory cell indexed by register r_j the contents of register r_i.
2. **Computation**: RAM machines can carry out computation on registers such as arithmetic operations, logical operations, and comparisons.
3. **Control flow**: As in the case of Turing machines, the chose of what instruction to perform next can depend on the state of the RAM machine, which is captured by the contents of its register.

Just as the NAND-TM programming language models Turing machines, we can also define a **NAND-RAM programming language** that models RAM machines. The NAND-RAM programming language extends NAND-TM by adding the following features:

1. The variables of NAND-RAM are allowed to be (non-negative) *integer valued* rather than only Boolean. That is, a scalar variable `foo` holds a nonnegative integer in $\mathbb{N}$ and an array variable `Bar` holds an array of integers. As in the case of RAM machines, we will not allow integers of unbounded size.
2. We allow *indexed access* to arrays. If `foo` is a scalar and `Bar` is an array, then `Bar[foo]` refers to the location of `Bar` indexed by the value of `foo`. Note that this means that we don't need to have a special index variable `i` anymore.

3. We will assume that for Boolean operations such as `NAND`, a zero valued integer is considered as *false*, and a nonzero valued integer is considered as *true*.
4. In addition to `NAND`, NAND-RAM also includes all the basic arithmetic operations of addition, subtraction, multiplication, integer division, as well as comparisions (equal, greater/less than, etc.).
5. NAND-RAM includes conditional statements `if/then` as a part of the language.
6. NAND-RAM contains looping constructs such as `while` and `do` as part of the language.

It is easy to see that NAND-RAM programs are clearly more powerful than NAND-TM, and so if a function F is computable by a NAND-TM program then it can be computed by a NAND-RAM program. It turns out to be true that if a function is computable by a NAND-RAM program, then it can also be computed by a NAND-TM program.

Theorem 5.2 ()

Turing machines (aka NAND-TM programs) and RAM machines (aka NAND-RAM programs) are equivalent. That is, for every function

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

F is computable by a NAND-TM program if and only if F is computable by a NAND-RAM program. Therefore, all four models are equivalent to one another.

6 Turing Completeness and Equivalence

Even though the notion of computing a function using Turing machines is crucial in theory, it is not a practical way of preforming computation. But in addition to defining computable functions with Turing machines, there are many equivalent conditions of computability under a wide variety of computational models. This notion is known as *Turing completeness* or *Turing equivalence*.

Any of the standard programming languages such as C, Java, Python, Pascal, Fortran, have very similar operations to NAND-RAM. Indeed, ultimately, they can all be executed by machines which have a fixed number of registers and a large memory array. Hence, with the equivalence theorem, we can simulate any program in such a programming language by a NAND-TM program. In the other direction, it is a fairly easy programming exercise to write an interpreter for NAND-TM in any of the above programming languages. Hence we can also simulate NAND-TM programs (and Turing machines) using these programming languages.

Definition 6.1 ()

A computational system is said to be **Turing-complete** or **computationally universal** if it can be be used to simulate any Turing machine or NAND-TM.
Very much related, the property of being *equivalent* in power to Turing machines/NAND-TM is called **Turing equivalent**. That is, two computer P and Q are equivalent if P can simulate Q and Q can simulate P. All known Turing complete systems are Turing equivalent.

The equivalence between Turing machines and RAM machines allows us to choose the most convenient language for the task at hand:

1. When we want to *prove a theorem* about all programs/algorithms, we can use Turing machines (or NAND-TM) since they are simpler and easier to analyze.
2. If we want to show that a certain function *cannot* be computed, then we will use Turing machines.
3. When we want to show that a function can be computed we can use RAM machines or NAND-RAM, because they are easier to program in and correspond more closely to high level programming languages

we are used to. In fact, we will often describe NAND-RAM programs in an informal manner, trusting that the reader can fill in the details and translate the high level description to the precise program. (This is just like the way people typically use informal or "pseudocode" descriptions of algorithms, trusting that their audience will know to translate these descriptions to code if needed.)

A formal definition of Turing completeness is as follows. This is also referred to as *Godel Numbering*, which is a function that assigns to each symbol and well-formed formula of some formal language a unique natural number, called its Gödel number

Definition 6.2 (Turing Completeness and Equivalence)

Let $\mathcal{F}$ be the set of all partial functions from $\{0,1\}^*$ to $\{0,1\}^*$. A **computational model** is a map

$$\mathcal{M} : \{0,1\}^* \longrightarrow \mathcal{F}$$

We say that a program $P \in \{0,1\}^*$ **$\mathcal{M}$-computes** a function $F \in \mathcal{F}$ if

$$\mathcal{M}(P) = F$$

A computational model $\mathcal{M}$ is **Turing complete** if there is a computable map

$$ENCODE_{\mathcal{M}} : \{0,1\}^* \longrightarrow \{0,1\}^*$$

such that for every Turing machine N (represented as a string), $\mathcal{M}(ENCODE_{\mathcal{M}}(N))$ is equal to the partial function computed by N.
A computational model $\mathcal{M}$ is **Turing equivalent** if it is Turing complete and there exists a computable map $DECODE_{\mathcal{M}} : \{0,1\}^* \longrightarrow \{0,1\}^*$ such that for every string $P \in \{0,1\}^*$, $N = DECODE_{\mathcal{M}}(P)$ is a string representation of a Turing machine that computes the function $\mathcal{M}(P)$.

6.1 Cellular Automata

Many physical systems can be described as consisting of a large number of elementary components that interact with one another. One way to model such systems is using cellular automata. This is a system that consists of a large (or even infinite) number of cells. Each cell only has a constant number of possible states. At each time step, a cell updates to a new state by applying some simple rule to the state of itself and its neighbors.

Definition 6.3 ()

An example of a cellular automaton is **Conway's Game of Life**. In this automata the cells are arranged in an infinite two dimensional grid. Each cell has only two states:

1. Dead: which we encode as a 0
2. Alive: which we encode as 1

The next state of a cell depends on its previous state and the states of its 8 adjacent neighbors, which can be modeled with a transition function

$$r : \Sigma^8 \longrightarrow \Sigma$$

A dead cell becomes alive only if exactly three of its neighbors are alive. A live cell continues to live if it has two or three live neighbors.

Even though the number of cells is potentially infinite, we can encode the state using a finite-length string by only keeping track of the live cells. If we initialize the system in a configuration with a finite number of live cells, then the number of live cells will stay finite in all future steps. Note that this is a discrete time Markov chain.

Since the cells in the game of life are arranged in an infinite two-dimensional grid, it is an example of a *two dimensional cellular automaton*. We can get even simpler by setting a *one dimensional cellular automaton*, where the cells are arranged in an infinite line.

Theorem 6.1 ()

Conway's Game of Life is Turing complete.

6.1.1 One-Dimensional Cellular Automata

Definition 6.4 ()

Let $\Sigma = \{0, 1, \emptyset\}$. A **one-dimensional cellular automaton** of alphabet Σ is described by a *transition rule*

$$r : \Sigma^3 \longrightarrow \Sigma$$

A **configuration** of the automaton r is a function $A : \mathbb{Z} \longrightarrow \Sigma$; that is, A just represents an infinite sequence of letters in the alphabet Σ. If an automaton with rule r is in configuration A, then its next configuration $A' = NEXT_r(A)$, is the function A' such that

$$A'(i) = r\big(A(i-1), A(i), A(i+1)\big)$$

In other words, the next state of the automaton r at point i is obtained by applying the rule r to the values of A at i and its two neighbors.
It is also said that a configuration of an automaton r is **finite** if there is only some finite number of indices $i_0, ..., i_{j-1}$ in $\mathbb{Z}$ such that $A(i_j) \neq \emptyset$.

If the alphabet is only $\{0, 1\}$, then there can be a total of $2^8 = 256$ total possible one dimensional cellular automata. For example, the cellular automaton with the transition rule

$$r(L, C, R) \equiv C + R + CR + LCR \pmod 2$$

can be expressed with the table (called rule 110)

111	110	101	100	011	010	001	000
0	1	1	0	1	1	1	0

However, many of them are trivially equivalent to each other up to a simple transformation of the underlying geometry, such as with reflections, translations, or rotations. This reduces the possible unique automata to 88, only one of which is Turing complete.

Theorem 6.2 ()

The Rule 110 cellular automaton is Turing complete. That is, any calculation or computer program can be simulated using this automaton.

Definition 6.5 (Configuration of Turing Machines)

Let M be a Turing machine with tape alphabet Σ and state space $[k]$. A **configuration** of M is a string

$$\alpha \in \overline{\Sigma}^*, \text{ where } \overline{\Sigma} = \Sigma \times \big(\{\cdot\} \cup [k]\big)$$

that satisfies that there is exactly one coordinate i for which $\alpha_i = (\sigma, s)$ for some $\sigma \in \Sigma$ and $s \in [k]$. For all other coordinates j, $\alpha_j = (\sigma', \cdot)$ for some $\sigma' \in \Sigma$. A configuration of $\alpha \in \overline{\Sigma}^*$ of M corresponds to the following staet of its execution:

1. M's tape contains $\alpha_{j,0}$ for all $j < |\alpha|$ and contains $\emptyset$ for all positions that are at least $|\alpha|$, where

we let $\alpha_{j,0}$ be the value σ such that $\alpha_j = (\sigma, t)$ with $\sigma \in \Sigma$ and $t \in \{\cdot\} \cup [k]$. In other words, since α_j is a pair of an alphebet symbol σ and either a state in $[k]$ or the symbol $\cdot$, $\alpha_{j,0}$ is the first component σ of this pair.
2. M's head is in the unique position i for which α_i has the form (σ, s) for $s \in [k]$, and M's state is equal to s.

Informally, a configuration can be interpreted simply as a string that encodes a *snapshot* of the Turing machine at a given point in the execution. It is also called a *core dump*. Such a snapshot must encode the following components:

1. The current head position.
2. The full contents of the large scale memory, that is the tape.
3. The contents of the "local registers," that is the state of the machine.

6.2 Lambda Calculus

The **Lambda calculus** is an abstract mathematical theory of computation, involving λ functions. It is a Turing complete language. λ calculus allows us to define "anonymous" functions. For example, instead of giving a name f to a function and defining it as

$$f(x) = x^2$$

we can write it anonymously (without naming it at all) as

$$x \mapsto x^2, \text{ or equivalently, } \lambda x.x^2$$

so $(\lambda x.x^2)(7) = 49$, or by dropping the parentheses, $(\lambda x.x^2)7 = 49$. That is, we can interpret $\lambda x.exp(x)$, where exp is some expression as a way of specifying the anonymous function $x \mapsto exp(x)$. This notation occurs in many programming languages, such as Python, where the squaring function is written `lambda x: x*x`.

Furthermore, in λ calculus functions are *first-class objects*, meaning that we can use functions as arguments to other functions. However, *all functions must take one input.*

Expressions can be thought of as programs in the language of lambda calculus. Given the notion of a variable, denoted by $x, y, z, ...$ we recursively define an expression inductively in terms of abstractions (anonymous functions) and applications as follows:

Definition 6.6 (λ expression)

Let Λ be the set of λ expressions. Then

1. Identifier: If x is a variable, then $x \in \Lambda$
2. Abstractions: If x is a variable and $\mathcal{M} \in \Lambda$, then $(\lambda x.\mathcal{M}) \in \Lambda$
3. Applications: If $\mathcal{M} \in \Lambda$ and $\mathcal{N} \in \Lambda$, then $\mathcal{M}\,\mathcal{N} \in \Lambda$
4. Grouping: If $\mathcal{M}$ is an expression, then $(\mathcal{M}) \in \Lambda$

Here are two important conventions:

1. Function application is left associative, unless stated otherwise by parentheses:

$$\mathcal{S}_1\mathcal{S}_2\mathcal{S}_3 \equiv \big((\mathcal{S}_1\mathcal{S}_2)\mathcal{S}_3\big)$$

2. Consecutive abstractions can be uncurried, e.g.

$$\lambda xyz.\mathcal{M} \equiv \lambda x.\lambda y.\lambda z.\mathcal{M}$$

3. The body of the abstraction extends to the right as far as possible

$$\lambda x.\mathcal{M}\,\mathcal{N} \equiv \lambda x.(\mathcal{M}\,\mathcal{N})$$

6.2.1 Applications

The notation for applying a function to a certain input is modeled by juxtaposition. That is,

$$f(a) \implies f\ a$$

where $f\ a$ means the function f *applied on input* a. However, since functions themselves could be inputs and outputs to other functions, we can use a method called **currying** to create multivariate functions. In the one below,

$$f\ a\ b \text{ , which stands for } f(a)(b)$$

this does not model a multivariate function f that takes two inputs. Rather, f takes one input a and outputs a function that takes one input b!

Example 6.1 ()

The addition function `add(a)(b)` can be modeled with 2 steps.

1. It takes the first argument a and outputs a function `adda` that takes another argument.

$$\texttt{add} : a \mapsto \texttt{adda}$$

2. `adda` takes argument b and adds b to the predetermined number a.

$$\texttt{adda} : b \mapsto a + b$$

Additionally, the expression

$$(f\ a)\ b \text{ , which stands for } \big(\text{f(a)}\big)\text{(b)}$$

is equivalent to $f\ a\ b$ since we have stated that function application is left associative. However,

$$f\ (a\ b) \text{ , which stands for f } \big(\text{a(b)}\big)$$

is a different expression, since now we are applying a onto b first, getting the output, and then applying f onto the output.

For example

$$((\lambda x.(\lambda y.x))2)9 = (\lambda y.2) = 9$$

Using a method called **currying**, we can actually create multivariate functions. For example, the function

$$\lambda x.(\lambda y.x + y)$$

maps x to the function $y \mapsto x + y$, which is equivalent to a function mapping $(x, y) \mapsto x + y$.

6.2.2 Abstractions

To understand abstractions, observe the four examples below (where $\implies$ means mapped to).

$$\begin{aligned} \lambda\ a.b &\quad a \implies b \\ \lambda\ a.b\ x &\quad a \implies b(x) \\ \lambda\ a.(b\ x) &\quad a \implies \big(b(x)\big) \\ (\lambda\ a.b)\ x &\quad (a \implies b)(x) \end{aligned}$$

In the second example, note that since the body of the abstraction extends to the far right as possible (i.e. the λ abstraction is greedy), it outputs the entire $b\ x$. The extra parentheses in the third line is not needed because of this convention. However, the parentheses in the fourth line is nontrivial. It says that $\lambda\ a.b$ outputs a function that acts on x. Finally, we are allowed to nest functions as such:

$$\lambda\ a.\lambda\ b.a \qquad a \implies b \implies a$$

The outermost λ takes in an a and returns a function that takes in a b, which in turn outputs the a. Note that $\lambda\ a.\lambda\ b.a = \lambda\ a.(\lambda\ b.a)$.

6.2.3 Beta Reduction

β-reduction refers to the process in simplifying a λ expression.

Example 6.2 ()

We can β reduce the expression into its simplest form, called the **beta normal form**.

$$\begin{aligned}((\lambda\, a.a)\ \lambda\, b.\lambda\, c.b)(x)\ \lambda\, e.f &= (\lambda\, b.\lambda\, c.b)(x)\ \lambda e.f \\ &= (\lambda\, c.x)\ \lambda\, e.f \\ &= x\end{aligned}$$

6.2.4 Combinators

Like transistors and Boolean gates, combinators are the atoms of more complicated functions in lambda calculus. We list five of them. Note that the cardinal can be build from other combinators.

Smyb	Bird	λ-Calculus	Use
I	Idiot	$\lambda\, a.a$	identity
M	Mockingbird	$\lambda\, f.ff$	self-application
K	Kestrel	$\lambda\, ab.a$	first, const
KI	Kite	$\lambda\, ab.b = KI = CK$	second
C	Cardinal	$\lambda\, fab.fba$	reverse arguments

6.2.5 Free and Bound Variables

In an abstraction like $\lambda x.x$, the variable x is something that has no original meaning but is a placeholder (i.e. it only has meaning within the λ function). We say that x is a variable **bound** to the λ. On the other hand, in $\lambda x.y$ i.e. a function which always returns y whatever it takes, y is a free variable since it has an independent meaning by itself. Because a variable is bound in some sub-expression does not mean it is bound everywhere. For example, the following is a valid expression (an example of application)

$$(\lambda x.x)(\lambda y.yx)$$

Here, the x in the second parenthesis has nothing to do with the one in the first. Formally,

Definition 6.7 ()

x is free...
1. in the expression x
2. in the expression $\lambda y.\mathcal{M}$ if $x \neq y$ and x is free in $\mathcal{M}$
3. in $\mathcal{M}\,\mathcal{N}$ if x is free in $\mathcal{M}$ or if it is free in $\mathcal{N}$

x bound...
1. in the expression $\lambda x.\mathcal{M}$
2. in $\mathcal{M}\,\mathcal{N}$ if x is bound in $\mathcal{M}$ or if it is bound in $\mathcal{N}$

Note that a variable can be both bound and free but they represent different things. An expression with no free variables is called a **closed expression**.

In addition, the concept of α equivalence states that any bound variable is a placeholder and can be replaced with a different variable, provided there are no clashes. A simple example is

$$\lambda x.x =_\alpha \lambda y.y$$

However,

$$\lambda x.(\lambda x.x) =_\alpha \lambda y.(\lambda x.x) \text{ but not to } \lambda y.(\lambda x.y)$$

Example 6.3 ()

The following λ expression can be simplified as such:

$$(\lambda x.(\lambda x.x))y =_\alpha \lambda y.y =_\alpha \lambda x.x$$

6.2.6 Booleans as Functions

Note that we can now define Booleans as functions! We can define a function f that outputs, one element if it is the True function and outputs another element if it is the False function. This can be done by defining:

$$T(a,b) = \lambda x.\lambda y.x(a)(b) = a \text{ (the Kestrel!)}$$
$$F(a,b) = \lambda x.\lambda y.y(a)(b) = b \text{ (the Kite!)}$$

Similarly, we can define the not function using the Cardinal.

Symb	Name	λ-Calculus	Use
T	True	$\lambda\, ab.a = K$	encoding for True
F	False	$\lambda\, ab.b$	encoding for False
	Not	$\lambda\, p.pFT = C$	negation

It is easy to see C as the negation function since

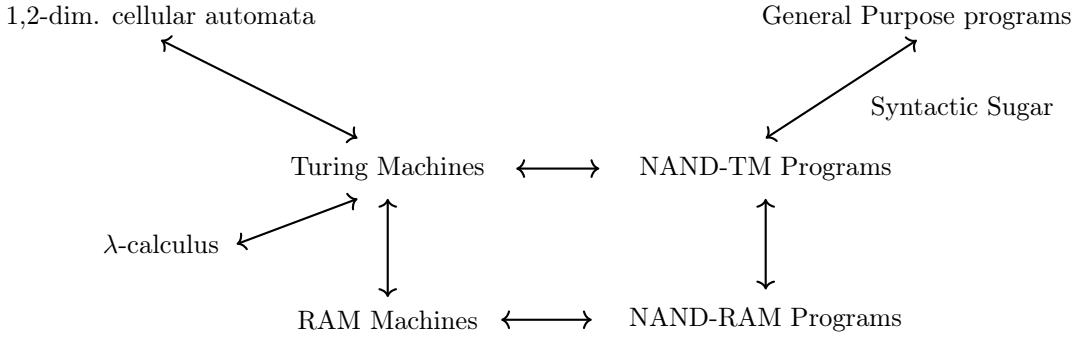
$$K(a)(b) = a \implies CK(a)(b) = b$$
$$KI(a)(b) = b \implies CKI(a)(b) = a$$

With this, we can build more complex logic gates, making the lambda calculus equivalent in computing power to NAND-CIRC programs. Similarly, we can cleverly implement recursion and arrays into this language, therefore making the lambda calculus Turing complete. To implement infinite loops, consider the λ expression

$$\lambda x.xx\ \lambda x.xx$$

If we try to simply this expression by invoking the left hand function on the right one, then we just get another copy of this expression.

The Turing equivalence of the computing models we have talked about can be visualized below:



7 Universality and Uncomputability

It turns out that uniform models such as Turing machines or NAND-TM programs allow us to obtain a truly *universal Turing machine* U that can evaluate all other machines, including machines that are more complex than U itself. Similarly, there is a *Universal NAND-TM program* U' that can evaluate all NAND-TM programs, including programs that have more lines than U'.

The existence of such a universal program/machine underlies the technological advances made up to now. Rather than producing special purpose calculating devices such as the abacus, the slide ruler, and machines that compute various trigonometric series, this universal property allows us to build a machine that, via software, can be extended to do arbitrary computations, i.e. a *general purpose computer*.

Theorem 7.1 (Universal Turing Machine)

There exists a Turing machine U such that on every string M which represents a Turing machine and $x \in \{0,1\}^*$,

$$U(M,x) = M(x)$$

That is, if the machine M halts on x and outputs some $y \in \{0,1\}^*$, then $U(M,x) = y$ and if M does not halt on x (i.e. $M(x) = \perp$), then $U(M,x) = \perp$.

There is more than one Turing machine U that satisfies the theorem above.

Definition 7.1 (String representation of Turing machine)

Let M be a Turing machine with k states and size l alphabet

$$\Sigma = \{\sigma_0, \sigma_1, ..., \sigma_{l-1}\}$$

(We use the convention $\sigma_0 = 0, \sigma_1 = 1, \sigma_2 = \emptyset, \sigma_3 = \triangleright$. We represent M as the triple (k,l,T), where T is the table of valies for δ_M:

$$T = \big(\delta_M(0,\sigma_0), \delta_M(0,\sigma_1), ..., \delta_M(k-1,\sigma_{l-1})\big)$$

where each value $\delta_M(s,\sigma)$ is a triple (s',σ',d) with $s' \in [k], \sigma' \in \Sigma$, and d a number in $\{0,1,2,3\}$ encoding one of $\{\mathsf{L},\mathsf{R},\mathsf{S},\mathsf{H}\}$. Thus, such a machine M is encoded by a list of $2+3k\cdot l$ natural numbers. The **string representation** of M is obtained by concatenating prefix-free representations of all these integers. If a string $\alpha \in \{0,1\}^*$ does not represent a list of integers in the form above, then we treat it as representing the trivial Turing machine with one state that immediately halts on every input.

The big takeways so far are:

1. We can represent every Turing machine as a string.
2. Given the string representation of a Turing machine M and an input x, we can simulate M's execution on the input x. That is, if we want to simulate a new Turing machine M, we do not need to build a new physical machine, but rather can represent M as a string (i.e. using code) and then input M to the universal machine U.

7.1 Uncomputable Functions

Even though NAND-CIRC programs can compute every finite function $f : \{0,1\}^n \longrightarrow \{0,1\}$, NAND-TM programs can *not* compute every function $F : \{0,1\}^* \longrightarrow \{0,1\}$. That is, there exists such a function that is *uncomputable*!

Definition 7.2 ()

Let $HALT : \{0,1\}^* \longrightarrow \{0,1\}$ be the function such that for every string $M \in \{0,1\}^*$, $HALT(M,x) = 1$ if Turing machine M halts on the input x and $HALT(M,x) = 0$ otherwise.

Theorem 7.2 ()

The $HALT$ function is not computable. This leads to many other functions also being uncomputable.

It is surprising that such a simple program is actually uncomputable. That is, there is no *general procedure* that would determine for an *arbitrary* program P whether it halts or not.

7.2 Impossibility of General Software Verification

Definition 7.3 ()

Let there be a program P that computes a function. A **semantic property** or **semantic specification** of a program means properties of the *function* that the program computes, as opposed to the properties that depend on the particular syntax/code used by the program.

Example 7.1 ()

A semantic property of a program P is the property that whenever P is given an input string with an even number of 1's, it outputs 0. Another example is the property that P will always halt whenever the input ends with a 1.
In contrast the property that a C program contains a comment before every function declaration is not a semantic property, since it depends on the actual source code as opposed to the input/output relation.

Example 7.2 ()

Consider the following two C programs:

```c
int First(int n) {
    if (n<0) return 0;
    return 2*n;
}

int Second(int n) {
    int i = 0;
    int j = 0
    if (n<0) return 0;
    while (j<n) {
        i = i + 2;
        j= j + 1;
    }
    return i;
}
```

`First` and `Second` are two distinct C programs, but they compute the same function. Therefore, a *semantic property* would either be true for both programs or false for both, since it depends on the function the programs compute. One example of a semantic property is: *The program P computes a function f mapping integers to integers satisfying that $f(n) \geq n$ for every input n.*
A property is *not semantic* if it depends on the source code rather than the input/output behavior. An example of this would be: *The program contains the variable* `k` or *the program uses the* `while` *operation.*

Definition 7.4 (Semantic properties)

A pair of Turing machines M and M' are **functionally equivalent** if for every $x \in \{0,1\}^*$, $M(x) = M'(x)$ (including when the function outputs $\perp$).
A function $F : \{0,1\}^* \longrightarrow \{0,1\}$ is **semantic** if for every pair of strings M, M' that represent

functionally equivalent Turing machines, $F(M) = F(M')$. Note that we assume that every string represents *some* Turing machine.

We now present a theorem concerning the Halting problem (the problem of determining whether a Turing machine will halt or not on any arbitrary input). The Halting problem also turns out to be a linchpin of uncomputability.

Theorem 7.3 (Rice's Theorem)

Let $F : \{0,1\}^* \longrightarrow \{0,1\}$. If F is semantic and nontrivial, then it is uncomputable.

Corollary 7.1 ()

The following function is uncomputable:

$$COMPUTES - PARITY(P) = \begin{cases} 1 & P \text{ computes the parity function} \\ 0 & \text{else} \end{cases}$$

Therefore, we can see that the set $\mathbf{R}$ of computable Boolean functions is a proper subset of the set of all functions mapping $\{0,1\}^* \longrightarrow \{0,1\}$.

7.3 Context Free Grammars

When a person designs a programming language, they need to determine its *syntax*. That is, the designer decides which strings correspond to valid programs, and which ones do not (i.e. which strings contain a syntax error). To ensure that a compiler or interpreter always halts when checking for syntax errors, language designers typically *do not* use a general Turing-complete mechanism to express their syntax. Rather, they use a *restricted* computational model, most often being *context free grammars*.

Consider the function $ARITH : \Sigma^* \longrightarrow \{0,1\}$ that takes as input a string x over alphabet

$$\Sigma = \{(,), +, -, \times, \div, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

and returns 1 if and only if the string x represents a valid arithmetic expression. Intuitively, we build expressions by applying an operation such as $+, -, \times, \div$ to smaller expressions or enclosing them in parentheses. More precisely, we can make the following definitions:

1. A *digit* is one of the symbols $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.
2. A *number* is a sequence of digits (we will drop the condition that the sequence does not have a leading zero)
3. An *operation* is one of $+, -, \times, \div$.
4. An *expression* has either the form
   (a) *"number"*
   (b) *"sub-expression1 operation sub-expression2*
   (c) *"(sub-expression1)"*

   where "sub-expression1" and "sub-expression2" are themselves expressions. Note that this is a recursive function.

A context free grammar (CFG) is a formal way of specifying such conditions, consisting of a set of ruels that tell us how to generate strings from smaller components.

Definition 7.5 (Context Free Grammar)

Let Σ be some finite set. A **context free grammar (CFG) over** Σ is a triple (V, R, s) such that:

1. V, known as the *variables*, is a set disjoint from Σ
2. $s \in V$ is known as the *initial variable*
3. R is a set of *rules*. Each rule is a pair (v, z) with $v \in V$ and $z \in (\Sigma \cup V)^*$. We often write the rule (v, z) as
$$v \implies z$$
and say that the string z *can be derived* from the variable v.

Example 7.3 ()

The example of well-formed arithmetic expressions can be captured formally by the following context free grammar.

1. The alphabet Σ is $\{(,), +, -, \times, \div, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
2. The variables are $V = \{expression, number, digit, operation\}$
3. The rules are the set R containing the following 19 rules:
   (a) 4 Rules: $operation \implies +$, $operation \implies -$, $operation \implies \times$, $operation \implies \div$
   (b) 10 Rules: $digit \implies 0$, $digit \implies 1$, ..., $digit \implies 9$
   (c) Rule: $number \implies digit$
   (d) Rule: $number \implies digitnumber$
   (e) Rule: $expression \implies number$
   (f) Rule: $expression \implies expression\ operation\ expression$
   (g) Rule: $expression \implies (expression)$
4. The starting variable is $expression$.

8 Introduction

Up until now, we have been concerned with which functions are computable and which ones are not. But now we will address the finer question of the *time* that it takes to compute functions, as a function of their input length. Time complexity is extremely important to both the theory and practice of computing.

Note that the running time of an algorithm is *not* a number. It is a *function* of the length of the input. Informally, we describe *efficient algorithms* as ones that have computational complexity of $O(n^c)$ for a small constant c. For some problems we know efficient algorithms and for others the best known algorithms are exponential. It is also interesting that seemingly minor changes in a problem formulation can make the (known) complexity of a problem "jump" from polynomial to exponential.

Furthermore, the difference between polynomial vs exponential time is typically *insensitive* to the choice of the particular computational model: a polynomial-time algorithm is still polynomial whether you use Turing machines, RAM machines, or parallel cluster, and similarly an exponential-time algorithm will remain exponential in all of these platforms.

8.1 Finding the shortest path in a graph

The *shortest path problem* is the task of finding, given a graph $G = (V, E)$ and two vertices $s, t \in V$, the length of the shortest path between s and t (if such a path exists). That is, we want to find the smallest number k such that there are vertices $v_0, v_1, ..., v_k$ with $v_0 = s, v_k = t$ and for every $i \in \{0, ..., k-1\}$ an edge between v_i and v_{i+1}. Formally, we define $MINIPATH : \{0,1\}^* \longrightarrow \{0,1\}^*$ to be the function that on input a triple (G, s, t) (represented as a string) outputs the number k which the length of the shortest path in G between s and t or a string representing `no path` if no such path exists. This algorithm can also yield the actual path itself as a byproduct.

If each vertex has at least two neighbors, then there can be an exponential number of paths from s to t, but fortunately we do not have to enumerate them all to find the shortest path. We can find the shortest path using a breadth first search (BFS), enumerating s's neighbors, and then neighbors' neighbors, etc.. in order. If we maintain the neighbors in a list we can perform a BFS in $O(n^2)$ time, while using a *queue* we can do this in $O(m)$ time. Dijkstra's algorithm is a well-known generalization of BFS to *weighed graphs*, where each edge is given a numerical weight (e.g. the distance between two nodes).

8.1.1 Finding the longest path in a graph

The *longest path problem* is the task of finding the length of the *longest* simple (i.e., non-intersecting) path between a given pair of vertices s and t in a given graph G. In particular, finding the longest path is a generalization of the famous *Hamiltonian path problem* which asks for a maximally long simple path (i.e., path that visits all n vertices once) between s and t, as well as the notorious *traveling salesman problem (TSP)* of finding (in a weighted graph) a path visiting all vertices of cost at most w. TSP is a classical optimization problem, with applications ranging from planning and logistics to DNA sequencing and astronomy.

Surprisingly, while we can find the shortest path in $O(m)$ time, there is no known algorithm for the longest path problem that significantly improves on the trivial "exhaustive search" or "brute force" algorithm that enumerates all the exponentially many possibilities for such paths. Specifically, the best known algorithms for the longest path problem take $O(c^n)$ time for some constant $c > 1$. Currently the best record is $c \approx 1.65$.

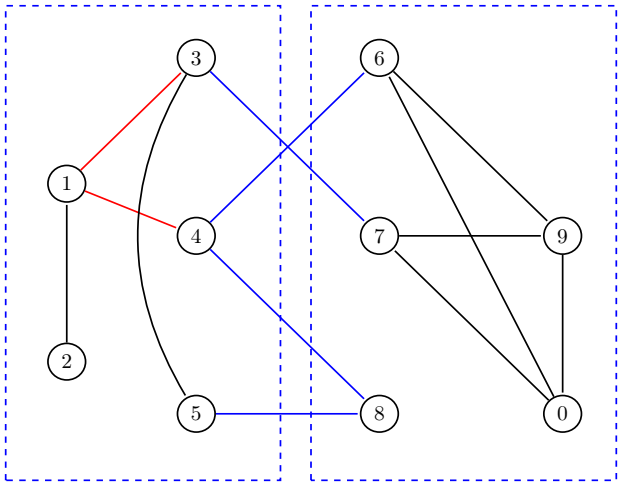
8.1.2 Finding the minimum cut in a graph

Definition 8.1 ()

Given a graph $G = (V, E)$, a **cut** of G is a subset $S \subset V$ such that S is neither empty not is it all of V. The edges cut by S are those edges where one of their endpoints is in S and the other is in $\overline{S} = V \setminus S$. We denote this set of edges by $E(S, \overline{S})$. If $s, t \in V$ are a pair of vertices, then an s, t **cut** is a cut such that $s \in S$ and $t \in \overline{S}$.

The *minimum s, t cut problem* is the task of finding, given s, t, the minimum number k such that there is an s, t cut cutting k edges. This also yields the set S that achieves this minimum. Formally, we define $MINCUT : \{0,1\}^* \longrightarrow \{0,1\}^*$ to be the function that on input a string representing a triple $(G = (V, E), st)$ of a graph and two vertices, outputs the minimum number k such that there exists a set $S \subset V$ with $s \in S, t \notin S$, and $|E(S, \overline{S})| = k$.

In the diagram below, an example of a cut is labeled with blue, while the minimum $1, 0$ cut is labeled in red.



There are many applications to computing minimum s, t cuts since minimum cuts often correspond to *bottlenecks*. The applications in communication or railroad networks is obvious now. Additionally, in the

setting of image segmentation, one can define a graph whose vertices are pixels and whose edges correspond to neighboring pixels of distinct colors. If we want to separate the foreground from the background, then we can pick (or guess0 a foreground pixel s and a background pixel t and ask for a minimum cut between them.

The naive algorithm for computing $MINCUT$ will check all 2^n possible subset of an n-vertex graph, but we can actually build algorithm that compute $MINCUT$ in polynomial time.

8.1.3 Min-Cut Max-Flow and Linear Programming

We can obtain a polynomial-time algorithm for computing $MINCUT$ using the *Max-Flow Min-Cut Theorem*.

Theorem 8.1 (Max-Flow Min-Cut Theorem)

In a *flow network* G (we can just interpret this as a weighted directed graph), the maximum amount of flow passing from source $s \in V$ to sink $t \in V$ is equal to the total weight of the edges in a minimum cut. If the graph is unweighted (i.e. every edge has unit capacity), then the maximum flow is just equal to the minimum cut k. The **maximum s,t flow** is the maximum units of water that we could transfer from s to t over these pipes. If there is an s,t cut of k edges, then the maximum flow is at most k.

It is easy to see why this theorem is when we interpret the minimum cut S acting as a bottleneck that restricts the flow the most. The Max-Flow Min-Cut Theorem reduces the task of computing a minimum cut of the task of computing a *maximum flow*. For this problem, the *Ford-Fulkerson Algorithm* is direct way to compute such a flow using incremental improvements. This is a special case of a more more general tool known as *linear programming*.

Definition 8.2 ()

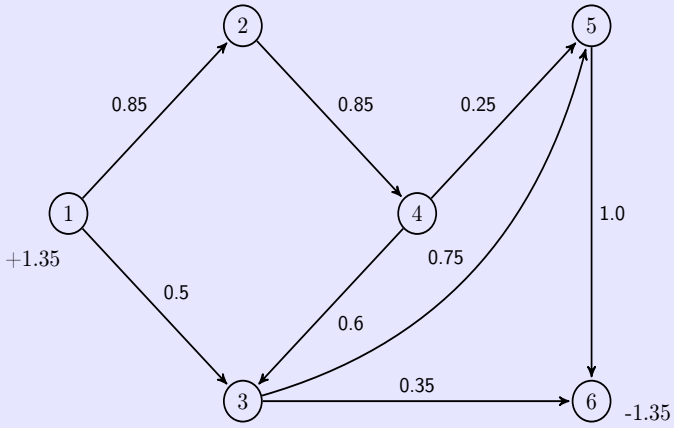
A **flow** on a graph G of m edges represents the weight of each edge, which can be interpreted as the amount of water per time-unit that flows through each edge. The flow on this graph of m edges can be modeled as a vector $x \in \mathbb{R}^m$ where for every edge e, x_e corresponds to the amount of water per time-unit that flows on e. We think of an edge e as an ordered pair (u, v) (can be chosen arbitrarily) and let x_e be the amount of flow that goes from u to v. Since every edge has capacity one, $-1 \leq x_e \leq 1$ for every edge e. Finally, a valid flow has the property that the amount of water leaving the source s is the same as the amount entering the sink t, and that for every other vertex v, the amount of water entering and leaving v is the same. Mathematically, we can write these properties as follows:

$$\begin{aligned} &\sum_s x_s + \sum_t x_t = 0 \\ &\sum_e x_e = 0 \ \ \forall\, v \in V \setminus \{s,t\} \\ &-1 \leq x_e \leq 1 \qquad\qquad\qquad\qquad \forall\, e \in E \end{aligned}$$

We write the source and sinks as summations since there may be multiple sources and sinks.

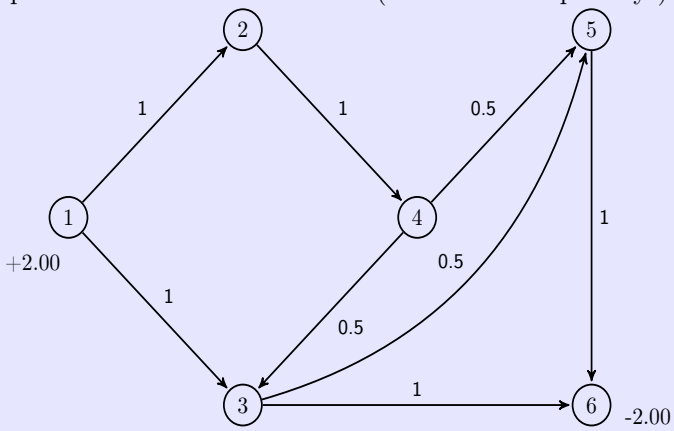
Example 8.1 ()

An example of such a viable sink is:



Note that the water flowing from the source and into the sink are both 1.35, and at each node, the water flowing in is equal to the water flowing out. The water flowing through each pipe is also less than 1.

From this very simple graph, we can see that the minimum $1, 6$ cut is $k = 2$, and therefore the maximum flow of water from node 1 to node 6 is 2 units of water per given time-interval. We can even construct this explicit "maximum flow" as such (there are multiple ways):



By generalizing this process, the maximum flow problem can be thought of as the task of maximizing $\sum_s x_s$ over all the vectors $x \in \mathbb{R}^m$ satisfying the properties above for graphs. Clearly, the function that maps $l : x \to x_s$ is linear, and maximizing this linear function $l(x)$ over the set $x \in \mathbb{R}^m$ that satisfy certain linear equalities and inequalities is known as *linear programming*. There are polynomial-time algorithms for solving linear programming, and hence we can solve the maximum flow (and so, minimum cut) problem in polynomial time. In fact, there are much better algorithms for maximum flow/minimum-cut, even for weighted directed graphs, with the record standing at $O(\min\{m^{10/7}, m\sqrt{n}\})$ time.

Definition 8.3 ()

Given a graph $G = (V, E)$, the ***global* minimum cut** of G is the minimum over *all* $S \subset V$ with $S \neq \emptyset, V$ of the number of edges cut by S. Therefore, the points s, t are not chosen initially, and every graph has a global minimum cut.

Theorem 8.2 ()

There is a polynomial-time algorithm to compute the global minimum cut of a graph.

Similarly, the *maximum cut* problem is the task of finding, given an input graph $G = (V, E)$, the subset $S \subset V$

that *maximizes* the number of edges cut by S. This can also be defined for the s,t cut, too. But unlike the minimum cut problem which can be solved using a polynomial time-algorithm, there is no known algorithm solving maximum cut much faster than the trivial "brute force" algorithm that tries all 2^n possibilities for the set S.

8.1.4 Convexity

There is an underlying reason for the difference between the difficulty of maximizing and minimizing a function over a domain. If $D \subset \mathbb{R}^m$, then a function $f : D \longrightarrow R$ is *convex* if for every $x, y \in D$ and $p \in [0, 1]$,

$$f(px + (1-p)y) \leq pf(x) + (1-p)f(y)$$

Theorem 8.3 ()

Given a convex set $D \subset \mathbb{R}^m$ and convex function $f : D \longrightarrow R$, if x is a local minimum of f, then it is also a global minimum.

Proof.

Assume that x is the local minimum and there is a global minimum $y \neq x$. $f(y) < f(x)$, so every point $z = px + (1-p)y$ on the line segment between x and y will satisfy

$$f(z) \leq pf(x) + (1-p)f(y) < f(x)$$

and hence in particular x cannot be a local minimum.

In general, local minima of functions are much easier to find than global ones (e.g. using algorithms like gradient descent). Indeed, under certain technical conditions, we can often efficiently find the minimum of convex functions over a convex domain, and this is the reason why problems such as minimum cut and shortest path are easy to solve. On the other hand, maximizing a convex function over a convex domain (or equivalently, minimizing a concave function) can often be a hard computational task. A linear function is both convex and concave, which is the reason that both the maximization and minimization problems for linear functions can be done efficiently.

The minimum cut problem is not a priori a convex minimization task, because the set of potential cuts is discrete and not continuous. However, it turns out that we can embed it in a continuous and convex set via the (linear) maximum flow problem. The "max flow min cut" theorem ensures that this embedding is "tight" in the sense that the minimum "fractional cut" that we obtain through the maximum-flow linear program will be the same as the true minimum cut. Unfortunately, we don't know of such a tight embedding in the setting of the maximum cut problem.

8.2 Computational Problems Beyond Graphs

8.2.1 SAT

A **propositional formula** φ involves n variables $x_1, x_2, ..., x_n$ and the logical operators AND ($\wedge$), OR ($\vee$), and NOT ($\neg$, also denoted with a bar).

Definition 8.4 ()

We say that a propositional formula is in *conjunctive normal form (CNF)* if it is an AND of ORs or their negations. A term of the form x_i or $\overline{x_i}$ is called a **literal**.
Furthermore, we say that a formula is a **k-CNF** if it is an AND of ORs where each OR involves exactly k literals.

Example 8.2 ()

This is a CNF formula since it is an AND of ORs of literals.

$$(x_7 \vee \overline{x_{22}} \vee x_{15}) \wedge (x_{37} \vee x_{22}) \wedge (x_{55} \vee \overline{x_7})$$

Definition 8.5 ()

A **satisfying assignment** for CNF formula φ is a string $x \in \{0,1\}^*$ such that φ evaluates to *True* if we assign its variables the values of x.

Following this, the **satisfiability problem** is the task of determining, given a CNF formula φ, whether or not there exists a satisfying assignment for φ. More specifically, the k-SAT problem is the restriction of the satisfiability problem for the case that the input formula is a k-CNF.

Example 8.3 ()

The CNF formula

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge \overline{x_1}$$

is satisfiable by assigning $x = (x_1, x_2, x_3) = (FALSE, FALSE, arbitrary)$, since

$$\begin{aligned} &(FALSE \vee \overline{FALSE}) \wedge (\overline{FALSE} \vee FALSE \vee x_3) \wedge \overline{FALSE} \\ =&(FALSE \vee TRUE) \wedge (TRUE \vee FALSE \vee x_3) \wedge TRUE \\ =&TRUE \wedge TRUE \wedge TRUE \\ =&TRUE \end{aligned}$$

However, the CNF formula

$$x_1 \wedge \overline{x_1}$$

is not satisfiable, since neither $x_1 = TRUE$ nor $x_1 = FALSE$ will reduce the above statement to TRUE.

The trivial, brute-force algorithm for **2SAT** will enumerate all the 2^n assignments $x \in \{0,1\}^n$ but fortunately, we can do much better. Let us assume that there exists a satisfiable solution to this 2-CNF formula. Then, we can think of every constraint $l_i \vee l_j$ (where l_i, l_j are literals, corresponding to variables or their negations) as an *implication* $\overline{l_i} \implies l_j$, since if l_i is false then l_j must be true. Therefore, we can make a directed graph of the $2n$ literals $(x_1, ..., x_n, \overline{x_1}, ..., \overline{x_n})$ with every constraint $l_i \vee l_j$ corresponding to the directed edge $\overline{l_i} \to l_j$. With this, it can be shown that φ is unsatisfiable if and only if there is a variable x_i such that there is a directed path from x_i to $\overline{x_i}$ and from $\overline{x_i}$ to x_i (since this means that $x_i \implies ... \implies \overline{x_i}$, reaching a contradiction).

The **3SAT** problem is the task of determining satisfiability for 3-CNFs, and we do not know of a significantly better than brute force algorithm for 3SAT. The best known algorithms take roughly 1.3^n steps.

8.2.2 Solving Linear and Quadratic Equations

The standard Gaussian elimination algorithm can be used to solve a linear system of n equations in n variables in polynomial time. In fact, if we are willing to allow some loss in precision, there are algorithms that can handle linear *inequalities*, also known as linear programming. In contrast, if we would like *integer solutions*, the ask for solving linear equalities or inequalities is known as *integer programming*, and the best known algorithms are exponential time in the worst case.

However, if we would like to solve not just linear but equations involving quadratic terms of the form

$$a_{i,j} x_j x_k$$

That is, suppose that we are given a set of quadratic polynomials $p_1, ..., p_m$ and consider the homoegeneous equations $p_i(x) = 0$. To avoid issues with bit representations, we will always assume that the equations contain the constraints $x_i^2 - x_i = 0$ (with only solutions being $x_i = 0, 1$). This means that we can restrict attention to solutions in $\{0,1\}^n$. For this problem, we do not know a much better algorithm for this problem than the one that enumerates over all the 2^n possibilities.

8.2.3 Determinant and Permanent of a Matrix

Using the LUP decomposition algorithm (which is really dependent on polynomial-time Gaussian elimination), the determinant of an $n \times n$ matrix can be computed in polynomial time of arithmetic operations.

Definition 8.6 ()

The **permanent** of $n \times n$ matrix A is defined as

$$\text{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} A_{i,\sigma(i)}$$

That is, $\text{perm}(A)$ is defined analogously to the determinant except that we drop the sign of σ.

It turns out that we can find compute a function $\text{perm}_2(A)$ that computes the permanent modulo 2 in polynomial time, but as soon as we reach permanent modulo 3 or greater prime numbers, we do not know of a much better than brute force algorithm to even compute the permanent modulo 3.

8.2.4 Finding a Zero-Sum Equilibrium

Definition 8.7 ()

A **zero sum game** is a game between two players where the payoff for one is the same as the penalty for the other. A zero sum game can be specified by a $n \times n$ matrix A, where if player chooses action i and player 2 chooses action j then player one gets $A_{i,j}$ and player 2 loses the same amount.

The famous *Min Max theorem* of linear algebra states that we if allow probabilistic or mixed strategies (where a player does not choose a single action but rather a *distribution* over actions), then it does not matter who plays first and the end result will be the same. Mathematically, the min max theorem is that if we let δ_n be the set of probability distributions over $[n]$ (i.e. δ_n is the set of all nonnegative column vectors in $\mathbb{R}^n$ whose entries sum up to 1), then

$$\max_{p \in \delta_n} \min_{q \in \delta_n} p^T A q = \min_{q \in \delta_n} \max_{p \in \delta_n} p^T A q$$

This value can be computed efficiently by a linear program.

8.2.5 Finding a Nash Equilibrium

For games that are not zero sum, where the payoff of one player does not necessarily equal the loss of the other, there is the notion of a *Nash equilibrium* for such games as well. However, unlike zero sum games, we do not know of an efficient algorithm for finding a Nash equilibrium given the description of a general (non zero-sum) game. In particular, this means that there are games for which natural strategies will take an exponential number of steps to converge to an equilibrium.

8.2.6 Primality Testing and Integer Factoring

In order to determine whether a number N is prime or not, we can try dividing it by all the numbers up to $\sqrt{N}$, but this is still quite terrible computationally. But fortunately, a *probabilistic* algorithm to determine whether a given number N is prime or composite in time $poly(n)$ for $n = \log N$.

On the contrary, no such algorithm that could efficiently find the factorization of N is known.

8.3 Current Knowledge

The difference between an exponential and polynomial time algorithms might seem merely "quantitative" but it is in fact extremely significant. As we've already seen, the brute force exponential time algorithm runs out of steam very very fast, and in practice there might not be much difference between a problem where the best algorithm is exponential and a problem that is not solvable at all. Thus the efficient algorithms we mentioned above are widely used and power many computer science applications. Moreover, a polynomial-time algorithm often arises out of significant insight to the problem at hand, whether it is the max-flow min-cut result, the solvability of the determinant, or the group theoretic structure that enables primality testing. Such insight can be useful regardless of its computational implications.

At the moment we do not know whether the "hard" problems are truly hard, or whether it is merely because we haven't yet found the right algorithms for them. However, we will now see that there are problems that do inherently require exponential time. We just don't know if any of the examples above fall into that category.

9 Modeling Running Time

When talking about running time, what we care about is the *scaling behavior* of the number of steps as the input size grows (as opposed to a fixed number).

9.1 Formally Defining Running Time

We can informally define what it means for a function $F : \{0,1\}^* \longrightarrow \{0,1\}^*$ to be *computable* in time $T(n)$ steps, where T is some function mapping the length n of the input to the number of computation steps allowed.

Definition 9.1 ()

Let $T : \mathbb{N} \longrightarrow \mathbb{N}$ be some function. We say that a function $F : \{0,1\}^* \longrightarrow \{0,1\}^*$ is **computable in $T(n)$ Turing Machine time (TM-time for short)** if there exists a Turing machine M such that for every sufficiently large n and every $x \in \{0,1\}^n$, the machine halts after executing at most $T(n)$ steps and outputs $F(x)$.
We define $TIME_{\mathsf{TM}}\big(T(n)\big)$ to be the set of Boolean functions ($\{0,1\}^* \longrightarrow \{0,1\}$) that are computable in $T(n)$ TM time. Note that $TIME_{\mathsf{TM}}\big(T(n)\big)$ is a class of *functions*, not machines.

With this, we can formally define what is means for function $F : \{0,1\}^* \longrightarrow \{0,1\}$ to be computable in time at most $T(n)$ where n is the size of the input. Furthermore, the property of considering only "sufficiently large" n's is not very important but it is convenient since it allows us to avoid dealing explicitly with uninteresting "edge cases." We have also defined computability with Boolean functions for simplicity, but we can generalize this further.

9.1.1 Polynomial and Exponential Time

Definition 9.2 ()

The two main time complexity classes are defined:

1. **Polynomial time**: A function $F : \{0,1\}^* \longrightarrow \{0,1\}$ is **computable in polynomial time** if it is in the class
$$\mathbf{P} = \bigcup_{c \in \{1,\dots m\}} TIME_{\mathsf{TM}}\big(n^c\big), \quad m \in \mathbb{N}$$
That is, $F \in \mathbf{P}$ if there is an algorithm to compute F that runs in time at most *polynomial* in the length of the input.
2. **Exponential time**: A function $F : \{0,1\}^* \longrightarrow \{0,1\}$ is **computable in exponential time**

if it is in the class

$$\mathbf{EXP} = \bigcup_{c\in\{1,\dots,m\}} TIME_{\mathsf{TM}}\big(2^{n^c}\big)$$

That is, $F \in \mathbf{EXP}$ if there is an algorithm to compute F that runs in time at most *exponential* in the length of the input.

Summarizing this, we say that $F \in \mathbf{P}$ if there is a polynomial $p : \mathbb{N} \longrightarrow \mathbb{R}$ and a Turing machine M such that for every $x \in \{0,1\}^*$, when given input x, the Turing machine halts within at most $p(|x|)$ steps and outputs $F(x)$.

We say that $F \in \mathbf{EXP}$ if there is a polynomial $p : \mathbb{N} \longrightarrow \mathbb{R}$ and a Turing machine M such that for every $x \in \{0,1\}^*$, when given input x, M halts within at most $2^{p(|x|)}$ steps and outputs $F(x)$.

Lemma 9.1 ()

Since exponential time is much larger than polynomial time,

$$\mathbf{P} \subset \mathbf{EXP}$$

Time complexity for the previous algorithms are as follows:

P	**EXP** (not known to be **P**)
Shortest path	Longest path
Min cut	Max cut
2SAT	3SAT
Linear eqs	Quad eqs
Zerosum	Nash
Determinant	Permanent
Primality	Factoring

Many technological developments are centered around these facts. For example, the exponential time complexity of factoring algorithms is what makes the RSA-encryption so secure. If a polynomial time algorithm for factoring were to be discovered, RSA-encryption would be rendered obsolete.

9.2 Modeling Running Time Using RAM Machines/NAND-RAM

Despite the theoretical elegance of Turing machines, RAM machines and NAND-RAM programs are much more closely related to actual computing architecture. For example, even a "merge sort" program cannot be implemented on a Turing machines in $O(n \log n)$ time. We can define running time with respect to NAND-RAM programs just as we did for Turing machines.

Definition 9.3 ()

Let $T : \mathbb{N} \longrightarrow \mathbb{N}$. We say that a function $F : \{0,1\}^* \longrightarrow \{0,1\}^*$ is **computable in T(n) RAM time (RAM-time for short)** if there exists a NAND-RAM program P such that for every sufficiently large n and every $x \in \{0,1\}^n$, when given input x, the program P halts after executing at most $T(n)$ lines and outputs $F(x)$.

We define $TIME_{\mathsf{RAM}}\big(T(n)\big)$ to be the set of Boolean functions $(\{0,1\}^* \longrightarrow \{0,1\})$ that are computable in $T(n)$ RAM time.

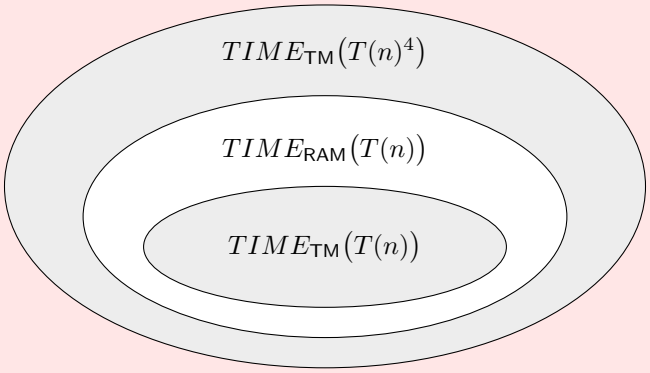
We will use $TIME\big(T(n)\big)$ to denote $TIME_{\mathsf{RAM}}\big(T(M)\big)$. However, as long as we only care about the difference between exponential and polynomial time, the model of running time we use does not make much difference. The reason is that Turing machines can simulate NAND-RAM programs with at most a polynomial overhead.

Theorem 9.1 (Relating RAM and Turing machines)

Let $T : \mathbb{N} \longrightarrow \mathbb{N}$ be a function such that $T(n) \geq n$ for every n and the map $n \mapsto T(n)$ can be computed by a Turing machine in time $O(T(n))$. Then,

$$TIME_{\mathsf{TM}}\big(T(n)\big) \subseteq TIME_{\mathsf{RAM}}\big(10 \cdot T(n)\big) \subseteq TIME_{\mathsf{TM}}\big(T(n)^4\big)$$

We can visually see this classification as



With this, we could have equally defined **P** as the class of functions computable by NAND-RAM programs (instead of Turing machines) that run in polynomial time in the length of the input. Similarly, with $T(n) = 2^{n^a}$, we see that the class **EXP** can also be defined as the set of functions computable by NAND-RAM programs in time at most $2^{p(n)}$ where p is some polynomial. This justifies the choice of **P** as capturing a technology-independent notion of tractability. Therefore, *all "reasonable" computational models are equivalent if we only care about the distinction between polynomial and exponential*, with reasonable referring to all scalable computational models that have been implemented except possibly quantum computers.

When considering general time bounds, we need to make sure to rule out some "exceptional" cases such as functions T that don't give enough time for the algorithm to even read the input, or functions where the time bound itself is uncomputable. More precisely, T must be a *nice function*.

Definition 9.4 ()

That is why we say that the function $T : \mathbb{N} \longrightarrow \mathbb{N}$ is a **nice time bound function** (**nice function** for short) if

1. for every $n \in \mathbb{N}$ $T(n) \geq n$ (T allows enough time to read the input)
2. for every $n' \geq n$, $T(n') \geq T(n)$ (T allows more time on longer inputs)
3. the map $F(x) = 1^{T(|x|)}$ (i.e. mapping a string of length n to a sequence of $T(n)$ ones) can be computed by a NAND-RAM program in $O(T(n))$ time

So, the following are examples of polynomially equivalent models:

1. Turing machines
2. NAND-RAM programs/RAM machines
3. All standard programming languages, including C/Python/Javascript...
4. The λ calculus
5. Cellular automata
6. Parallel computers
7. Biological computing devices such as DNA-based computers

The *Extended Church Turing Thesis* is the statement that this is true for all physically realizable computing models. In other words, the extended Church Turing thesis says that for every *scalable computing device* C (which has a finite description but can be in principle used to run computation on arbitrarily large inputs), there is some constant a such that for every function $F : \{0,1\}^* \longrightarrow \{0,1\}$ that C can compute on n length inputs using an $S(n)$ amount of physical resources. This is a strengthening of the plain Church Turing Thesis, which states that the set of computable functions is the same for all physically realizable models, but without requiring the overhead in the simulation between different models to be at most polynomial.

Like the Church-Turing thesis itself, the extended Church-Turing thesis is in the asymptotic setting and does not directly yield an experimentally testable prediction. However, it can be instantiated with more concrete bounds on the overhead, yielding experimentally- testable predictions such as the Physical Extended Church-Turing Thesis.

9.3 Efficient Universal Machine: A NAND-RAM Interpreter in NAND-RAM

We can now see that the universal Turing machine U, which can compute every Turing machine M, has a *polynomial* overhead for simulating a $NAND-TM$ program. That is, it can simulate T steps of a given $NAND-TM$ (or $NAND-RAM$) program P on an input x in $O(T^4)$ steps. But in fact, by directly simulating $NAND-RAM$ programs we can do better with only a *constant* multiplicative overhead.

Theorem 9.2 (Efficient Universality of NAND-RAM)

There exists a NAND-RAM program U satisfying the following:

1. *U is a universal NAND-RAM program*: For every NAND-RAM program P and input x, $U(P,x) = P(x)$ where by $U(P,x)$ we denote the output of U on a string encoding the pair (P,x).
2. *U is efficient*: There are some constants a, b such that for every $NAND-RAM$ program P, if P halts on input x after most T steps, then $U(P,x)$ halts after at most $C \cdot T$ steps where $C \leq a|P|^b$.

This leads to a corollary. Given any Turing machine M, input x, and *step budget* T, we can simulate the execution for M for T steps in time that is polynomial in T. Formally, we define a function $TIMEDEVAL$ that takes the three parameters M, x, and the time budget, and outputs $M(x)$ if M halts within at most T steps, and outputs 0 otherwise. That is, let $TIMEDEVAL : \{0,1\}^* \longrightarrow \{0,1\}^*$ be the function defined as

$$TIMEDEVAL(M, x, 1^T) = \begin{cases} M(x) & M \text{ halts within } \leq T \text{ steps on } x \\ 0 & \text{else} \end{cases}$$

Then, $TIMEDEVAL \in \mathbf{P}$, i.e. the timed universal Turing machine computes $TIMEDEVAL$ in polynomial time.

9.4 The Time Hierarchy Theorem

Some functions are uncomputable, but are there functions that can be computed, but only at an exorbitant cost? For example, is there a function that *can* be computed in time 2^n, but *cannot* be computed in time $2^{0.9n}$? It turns out that the answer is yes.

Theorem 9.3 (Time Hierarchy Theorem)

For every nice function $T : \mathbb{N} \longrightarrow \mathbb{N}$, there is a function $F : \{0,1\}^* \longrightarrow \{0,1\}$ in

$$TIME\big(T(n)\log n\big) \setminus TIME\big(T(n)\big)$$

There is nothing special about $\log n$. We could have used any other efficiently computable function that ends to infinity with n.

9.5 Non-Uniform Computation

10 Polynomial-Time Reductions

Let us redefine some of the problems into *decision problems*.

3SAT The *3SAT problem* can be phrased as the function $3SAT : \{0,1\}^* \longrightarrow \{0,1\}$ that takes as an input a 3CNF formula φ (i.e. a formula of the form $C_0 \wedge ... \wedge C_{m-1}$ where each C_i of the OR of three iterables) and maps φ to 1 if there exists some assignment to the variables of φ that causes it to evaluate to *true* and to 0 otherwise. For example,

$$3SAT\big((x_0 \vee \overline{x_1} \vee x_2) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_0} \vee \overline{x_2} \vee x_3)\big) = 1$$

since the assignment $x = 1101$ satisfies the input formula.

Quadratic Equations The *quadratic equations problem* corresponds to the function $QUADEQ : \{0,1\}^* \longrightarrow \{0,1\}$ that maps a set of quadratic equations E to 1 if there is an assignment x that satisfies all equations and to 0 otherwise.

Longest Path The *longest path problem* correpsonds to the function $LONGPATH : \{0,1\}^* \longrightarrow \{0,1\}^*$ that maps a graph G and a number k to 1 if there is a simple path in G of length at least k, and maps (G,k) to 0 otherwise.

Maximum Cut The *maximum cut problem* corresponds to the function $MAXCUT : \{0,1\}^* \longrightarrow \{0,1\}$ that maps a graph G and a number k to 1 if there is a cut in G that cuts at least k edges, and maps (G,k) to 0 otherwise.

All of these problems above are in **EXP** but it is not known whether or not they are in **P**. However, we can reduce these problems to ones that are in **P**, proving that they are indeed in **P**.

10.1 Polynomial-Time Reductions

Suppose that that $F, G : \{0,1\}^* \longrightarrow \{0,1\}$ are two Boolean functions. A *polynomial-time reduction* (or *reduction*) from F to G is a way to sho that F is "no harder" than G in the sense that a polynomial-time algorithm for G implies a polynomial-time algorithm for F.

Definition 10.1 (Polynomial-time reductions)

Let $F, G : \{0,1\}^* \longrightarrow \{0,1\}$. We say that **F reduces to G**, denoted by $F \leq_p G$, if there is a polynomial-time computable $R : \{0,1\}^* \longrightarrow \{0,1\}^*$ such that for every $x \in \{0,1\}^*$,

$$F(x) = G\big(R(x)\big)$$

We say that F and G have **equivalent complexity** if $F \leq_p G$ and $G \leq_p F$. Clearly, $\leq_p$ is a transitive property.

10.2 Reducing 3SAT to Zero-One and Quadratic Equations

Definition 10.2 ()

The **Zero-One Linear Equations problem** corresponds to the function

$$01EQ : \{0,1\}^* \longrightarrow \{0,1\}$$

whose input is a collection E of linear equations in variables $x_0, ..., x_{n-1}$, and the output is 1 iff there is an assignment $x \in \{0,1\}^n$ satisfying the matrix equation

$$Ax = b, \quad A \in \text{Mat}(m \times n, \{0,1\}), b \in \mathbb{N}^m$$

For example, if E is a string encoding the set of equations

$$x_0 + x_1 + x_2 = 2$$
$$x_0 + x_2 = 1$$
$$x_1 + x_2 = 2$$

then $01EQ(E) = 1$ since the assignment $x = 011$ satisfies all three equations.

Note that if we extended the field to $\mathbb{R}$, then this can be solved using Gaussian elimination in polynomial time, but there is no known efficiently algorithm to solve $01EQ$. This is stated in the following theorem.

Theorem 10.1 (Hardness of 01 Linear Equations)

$$3SAT \leq_p 01EQ$$

This means that finding an efficient algorithm to solve $01EQ$ would imply an algorithm for $3SAT$. We can further use this to reduce $3SAT$ to the quadratic equations problem, where $QUADEQ(p_0, ..., p_{m-1}) = 1$ if and only if there is a solution $x \in \mathbb{R}^n$ to the equations $p_i(x) = 0$ for $i = 0, ..., m-1$. For example, the following is a set of quadratic equations over the variables x_0, x_1, x_2:

$$x_0^2 - x_0 = 0$$
$$x_1^2 - x_1 = 0$$
$$x_2^2 - x_2 = 0$$
$$1 - x_0 - x_1 + x_0 x_1 = 0$$

Theorem 10.2 (Hardness of Quadratic Equations)

$$3SAT \leq_p QUADEQ$$

10.3 Independent Set and Other Graph Problems

Definition 10.3 ()

For a graph $G = (V, E)$, an **independent set**, also known as a **stable set**, is a subset $S \subseteq V$ such that there are no edges with both endpoints in S (in other words, $E(S, S) = \emptyset$). Trivially, every singleton (of one point) is an independent set.

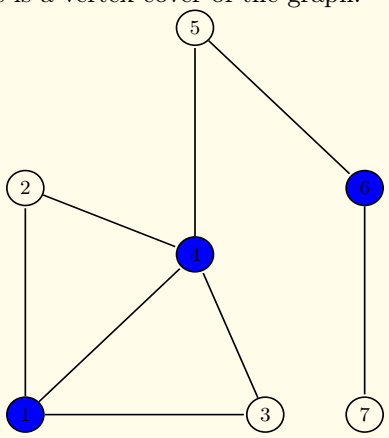
The **maximum independent set** problem is the task of finding the largest independent set in the graph. The independent set problem is naturally related to *scheduling problem*: if we put an edge between two conflicting tasks, then an independent set corresponds to a set of tasks that can all be scheduled together without conflicts.

Theorem 10.3 (Hardness of Independent Set)

$$3SAT \leq_p ISET$$

Definition 10.4 ()

A **vertex cover** in a graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices that touches all edges of G. For example, the following blue nodes is a vertex cover of the graph.



The **vertex cover problem** is the task to determine, given a graph G and a number k, whether there exists a vertex cover in the graph with at most k vertices. Formally, this is the function

$$VC : \{0,1\}^* \longrightarrow \{0,1\}$$

such that for every $G = (V, E)$ and $k \in \mathbb{N}$, $VC(G, k) = 1$ if and only if there exists a vertex cover $S \subset V$ such that $|S| \leq k$.

Theorem 10.4 ()

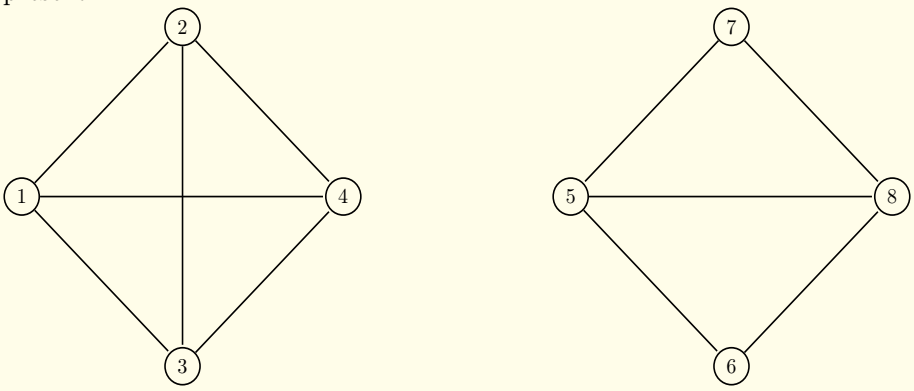
$$3SAT \leq_p VC$$

Definition 10.5 ()

A **clique** is a subset of vertices of an undirected graph such that every two distinct vertices in the graph are adjacent, i.e. connected by an edge.
The **maximum clique problem** corresponds to the function

$$CLIQUE : \{0,1\}^* \longrightarrow \{0,1\}$$

such that for a graph G and a number k, $CLIQUE(G, k) = 1$ iff there is a subset S of k vertices such that for *every* distinct $u, v \in S$, the edge u, v is in G. For example, in the graph below, the left subset of 4 vertices is indeed a clique, while the right subset of 4 is not since the edge connecting 6 to 7 is not present.



Theorem 10.5 ()

$$CLIQUE \leq_p ISET \text{ and } ISET \leq_p CLIQUE$$

Definition 10.6 ()

A **dominating set** in a graph $G = (V, E)$ is a subset $S \subset V$ of vertices such that for every $u \in V \setminus S$ is a neighbor in G

10.3.1 Anatomy of a Reduction

A reduction from problem F to a problem G is an algorithm that maps an input x for F to an input $R(x)$ for G. To show that the reduction is correct we need to show the properties of:

1. *efficiency*: algorithm R runs in polynomial time
2. *completeness*: if $F(x) = 1$, then $G(R(x)) = 1$
3. *soundness*: if $F(R(x)) = 1$, then $G(x) = 1$

Therefore, proving that problem G is a reduction of problem F is equivalent to showing the three properties above.

We finally reduce the 3SAT problem to the longest path problem.

Theorem 10.6 (Hardness of Longest Path)

$$3SAT \leq_p LONGPATH$$

That is, an efficient algorithm for the *longest path* problem would imply a polynomial-time algorithm for 3SAT. Therefore, we have shown that 3SAT is no harder than Quadratic Equations, Independent Set, Maximum Cut, and Longest Path.

11 NP, NP Completeness, and Cook-Levin Theorem

All of the problems that we have talked about are *search problems*, where the goal is to decide, given an instance x, whether there exists a solution y that satisfies some condition that can verified in polynomial time. For example, in 3SAT, the instance is a formula and the solution is an assignment to the variable; in Max-Cut the instance is a graph and the solution is a cut in the graph; and so on and so forth. It turns out that every such search problem can be reduced to 3SAT.

11.1 The Class NP

Intuitively, the class **NP** corresponds to the class of problems where it is *easy to verify* a solution (i.e. verification can be done by a polynomial-time algorithm). For example, finding a satisfying assignment to a 2SAT or 3SAT formula is such a problem, since if we are given an assignment to the variables of a 2SAT or 3SAT formula then we can efficiently verify that it satisfies all constraints.

That is, a Boolean function F is in **NP** if F has the form that on input string x, $F(x) = 1$ if and only if there exists a "solution" string w such that the pair (x, w) satisfies some polynomial-time checkable condition.

Definition 11.1 (NP - Nondeterministic Polynomial Time)

We say that $F : \{0,1\}^* \longrightarrow \{0,1\}$ is in **NP** if there exists some integer $a > 0$ and $V : \{0,1\}^* \longrightarrow \{0,1\}$ such that $V \in \mathbf{P}$ and for every $x \in \{0,1\}^n$,

$$F(x) = 1 \iff \text{there exists } w \in \{0,1\}^{n^a} s.t.\ V(xw) = 1$$

That is, for F to be in **NP**, there needs to exist some polynomial time computable verification function V such that if $F(x) = 1$, then there must exist w (of length polynomial in $|x|$) such that $V(xw) = 1$, and if $F(x) = 0$ then for *every* such w, $V(xw) = 0$. Since the existence of this string w certifies that $F(x) = 1$, w is often called the *certificate, witness*, or *proof* that $F(x) = 1$.

Some problems that are NP are:

1. $3SAT \in \mathbf{NP}$ since for every l-variable formula φ, $3SAT(\varphi) = 1$ if and only there exists a satisfying assignment $x \in \{0,1\}^l$ such that $\varphi(x) = 1$, and we can check this condition in polynomial time.
2. $QUADEQ \in \mathbf{NP}$ since for every l-variable instance of quadratic equations E, $QUADEQ(E) = 1$ if and only if there exists an assignment $x \in \{0,1\}^l$ that satisfies E. We can check the condition that x satisfies E in polynomial time by enumerating over all the equations in E, and for each such equation e, plug in the values of x and verify that e is satisfied.
3. $ISET \in \mathbf{NP}$ since for every graph G and integer k, $ISET(G,k) = 1$ if and only if there exists a set S of k vertices that contains no pair of neighbors in G. We can check the condition that S is an independent set of size $\geq k$ in polynomial time by first checking that $|S| \geq k$ and then enumerating over all edges $\{u,v\}$ in G, and for each such edge verify that either $u \neq S$ or $v \neq S$.
4. $LONGPATH \in \mathbf{NP}$ since for every graph G and integer k, $LONGPATH(G,k) = 1$ if and only if there exists a simple path P in G that is of length at least k. We can check the condition that P is a simple path of length k in polynomial time by checking that it has the form $(v_0, v_1, ..., v_k)$ where each v_i is a vertex in G, no v_i is repeated, and for every $i \in [k]$, the edge $\{v_i, v_{i+1}\}$ is present in the graph.
5. $MAXCUT \in \mathbf{NP}$ since for every graph G and integer k, $MAXCUT(G,k) = 1$ if and only if there exists a cut $(S, \overline{S})$ in G that cuts at least k edges. We can check that condition that $(S, \overline{S})$ is a cut of value at least k in polynomial time by checking that S is a subset of G's vertices and enumerating over all the edges $\{u,v\}$ of G, counting those edges such that $u \in S$ and $v \notin S$ or vice versa.

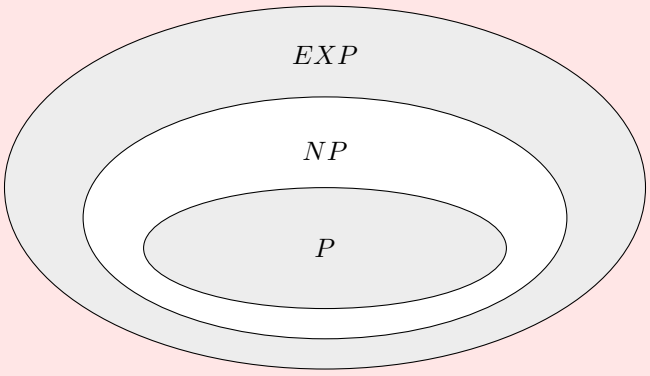
Theorem 11.1 ()

Verifying is no harder than solving:

$$\mathbf{P} \subseteq \mathbf{NP}$$

Furthermore,

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$$



Proof.

Suppose that $F \in \mathbf{P}$. Define the following function V:

$$V(x0^n) = \begin{cases} 1 & \text{iff } n = |x|, F(x) = 1 \\ 0 & \text{else} \end{cases}$$

Since $F \in \mathbf{P}$, we can clearly compute V in polynomial time as well. Let $x \in \{0,1\}^n$ be some string. If $F(x) = 1$ then $V(x0^n) = 1$. On the other hand, if $F(x) = 0$ then for every $w \in \{0,1\}^n$, $V(xw) = 0$. Therefore, setting $a = 1$ (i.e. $w \in \{0,1\}^{n^1}$), we see that V satisfies the NP condition.

11.2 NP Hard and NP Complete Problems

There are countless examples of problems for which we do not know if their best algorithm is polynomial or exponential, but we can show that they are in **NP**; that is, we don't know if they are easy to *solve*, but we do know that it is easy to *verify* a given solution. There are many other functions that we would like to compute that are easily shown to be in **NP**. In fact, it we can solve 3SAT then we can solve all of them!

Theorem 11.2 (Cook-Levin Theorem)

For every $F \in \mathbf{NP}$,

$$F \leq_p 3SAT$$

This immediately implies that $QUADEQ, LONGPATH$, and $MAXCUT$ (and really, *every* $F \in \mathbf{NP}$) all reduce to $3SAT$, meaning that all these problems are equivalent! All of these problems are the "hardest in **NP**" since an efficient algorithm for any one of them would imply an efficient algorithm for *all* the problems in **NP**.

Definition 11.2 ()

Let $G : \{0,1\}^* \longrightarrow \{0,1\}$. We say that G is **NP hard** if for every $F \in \mathbf{NP}$, $F \leq_p G$. We say that G is **NP complete** if G is **NP** hard and $G \in \mathbf{NP}$.

Therefore, despite their differences, 3SAT, quadratic equations, longest path, independent set, maximum cut, and thousands of other problems are all **NP** complete. Again, this means that *if a single **NP** complete problem has a polynomial-time algorithm, then there is such a polynomial-time algorithm for every decision problem that corresponds to the existence of an efficiently verifiable solution (i.e. is NP), which would imply that **P** = **NP**.*

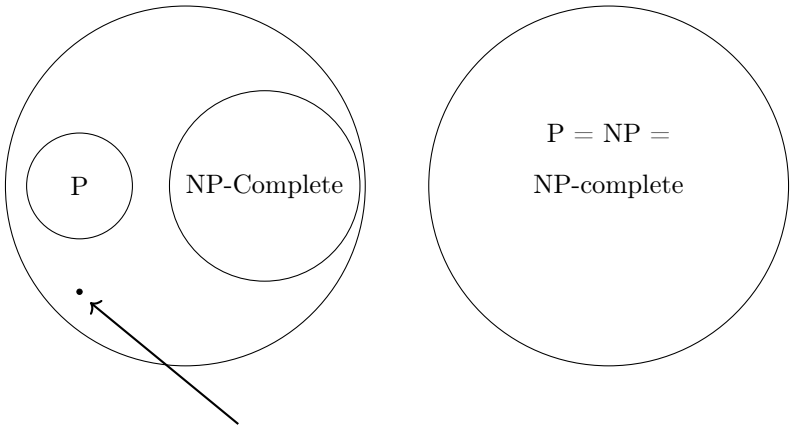
11.3 P = NP?

However, a polynomial-time algorithm for even a single one of the **NP** complete problems has even been found, proving support that $\mathbf{P} \neq \mathbf{NP}$

One of the mysteries of computation is that people have observed a certain empirical "zero-one law" or "dichotomy" in the computational complexity of natural problems, in the sense that many natural problems are either in **P** (often in $TIME(O(n))$ or $TIME(O(n^2))$), or they are **NP** hard. This is related to the fact that for most natural problems, the best known algorithm is either exponential or polynomial, rather than any strange function in between.

However, it is believed that there exist problems in **NP** that are neither in **P** nor are **NP** complete, and in fact a result known as **Lander's Theorem** shows that if $\mathbf{P} \neq \mathbf{NP}$, then this is indeed the case. Therefore, we are left with two cases:

1. If $\mathbf{P} \neq \mathbf{NP}$, meaning that $\mathbf{P}$ is a strict subset of $\mathbf{NP}$ and by Lander's theorem, $\mathbf{NP}$ complete problems do not cover all of $\mathbf{NP} \setminus \mathbf{P}$. (left)
2. If $\mathbf{P} = \mathbf{NP}$, meaning that $\mathbf{P} = \mathbf{NP} = \mathbf{NP}$ complete. (right)



11.4 NANDSAT, 3NAND Problems

Definition 11.3 ()

The function $NANDSAT : \{0,1\}^* \longrightarrow \{0,1\}$ is defined as follows:

1. The input to $NANDSAT$ is a string Q representing a NAND-CIRC program (or equivalently, a circuit with $NAND$ gates)
2. The output of $NANDSAT$ on input Q is 1 if and only if there exists a string $w \in \{0,1\}^n$ (where n is the number of inputs to Q) such that $Q(w) = 1$.

Definition 11.4 ()

The $3NAND$ problem is defined as follows:

1. The input is a logical formula Ψ on a set of variables $z_0, ..., z_{r-1}$ which is an AND of constraints of the form $z_i = NAND(z_j, z_k)$.
2. The output is 1 is and only if there is an input $z \in \{0,1\}^r$ that satisfies all of the constraints.

Example 11.1 ()

The following is a $3NAND$ formula with 5 variables and 3 constraints:

$$\Psi = \big(z_3 = NAND(z_0, z_2)\big) \wedge \big(z_1 = NAND(z_0, z_2)\big) \wedge \big(z_4 = NAND(z_3, z_1)\big)$$

In this case $3NAND(\Psi) = 1$, since the assignment $z = 01010$ satisfies it. Given a $3NAND$ formula Ψ of r variables and an assignment $z \in \{0,1\}^r$, we can check in polynomial time whether $\Psi(z) = 1$, and hence $3NAND \in \mathbf{NP}$.

Theorem 11.3 ()

$NANDSAT$ and $3NAND$ is $\mathbf{NP}$ complete.

12 Probabilistic Computation

It turns out that randomness can actually be a resource for computation, enabling us to achieve tasks much more efficiently than previously known. This advantage comes from the idea that calculating the statistics of a system could be done much faster by running several randomized simulations rather than explicit calculations, and these types of randomized algorithms are known as *Monte Carlo algorithms*.

12.1 Finding Approximately Good Maximum Cuts

Recall the maximum cut problem of finding, given a graph $G = (V, E)$, the cut that maximizes the number of edges. This problem is **NP**-hard, which means that we do not know of any efficient algorithm that can solve it, but randomization enables a simple algorithm that can cut at least half of the edges.

Theorem 12.1 (Approximating Max Cut)

There is an efficient probabilistic algorithm that on input an n-vertex m-edge graph G, outputs a cut $(S, \overline{S})$ that cuts at least $m/2$ of the edges of G in expectation.

Proof.

We simply choose a *random cut*: we choose a subset S of vertices by choosing every vertex v to be a member of S with probability $1/2$ independently. More specifically, upon input of a graph $G = (V, E)$ with vertices $(v_0, ..., v_{n-1})$, we do

1. Pick x uniformly at random in $\{0, 1\}^n$
2. Let $S \subseteq V$ be the set $\{v_i \mid x_i = 1, i \in [n]\}$ that includes all vertices corresponding to coordinates of x where $x_i = 1$.
3. Output the cut $(S, \overline{S})$.

We claim that the expected number of edges cut by the algorithm is $m/2$. Indeed, for every edge $e \in E$, let X_e be the random variable such that $X_e(x) = 1$ if the edge is cut by x, and let $X_e(x) = 0$ otherwise. It is not hard to see that the probability of $X_e(x) = 1$ is $\frac{1}{2}$ (when exactly one of the vertices are in S), and hence

$$\mathbb{E}(X_e) = 1/2$$

Summing this over all edges and by linearity of expectation, we get

$$\mathbb{E}(X) = \sum_{e \in E} \mathbb{E}(X_e) = m \cdot \frac{1}{2} = \frac{m}{2}$$

In fact, for *every graph* G, the algorithm is guaranteed to cut half of the edges of the input graph in expectation.

12.1.1 Amplifying the success of randomized algorithms

But note that expectation does not imply concentration. Luckily, we can *amplify* the probability of success by repeating the process several times and outputting the best cut we find. We assume that the probability that the algorithm above succeeds in cutting at least $m/2$ edges is not *too* tiny.

Lemma 12.1 ()

The probability that a random cut in an m edge graph cuts at least $m/2$ edges is at least $\frac{1}{2m}$.

Proof.

This is quite trivial when looking at specific cases. For example, take the case when $m = 1000$ edges. In this case, one can shot that we will cut at least 500 edges with probability at least 0.001 (and so in particular larger then $\frac{1}{2m} = \frac{1}{2000}$). Specifically, if we assume otherwise, then this means that with probability more than 0.999 the algorithm cuts 499 or fewer edges. But since we can never cut more than the total of 1000 edges, given this assumption, the highest value of the expected number of edges cut is if we cut exactly 499 edges with probability 0.999 and cut 1000 edges with probability 0.001. But this leads to the expectation being

$$0.999 \cdot 499 + 0.001 \cdot 1000 < 500$$

which contradicts the fact that the expectation to be at least 500 in the previous theorem. Generalizing this to m edges, we find that the expected number of edges cut is

$$pm + (1-p)\Big(\frac{m}{2} - \frac{1}{2}\Big) \leq pm + \frac{m}{2} - \frac{1}{2}$$

But since $p < \frac{1}{2m} \implies pm < 0.5$, the right hand side is smaller than $m/2$, contradicting the fact that the expected number of edges cut is at least $m/2$.

12.1.2 Success Amplification

To increase the chances of success, we simply need to repeat our program many times, with fresh randomness each time, and output the best cut we get in one of these repetitions. It turns our that if we repeat this experiment $2000m$ times, then by using the inequality

$$\Big(1 - \frac{1}{k}\Big)^k \leq \frac{1}{e} \leq \frac{1}{2}$$

we can show that the probability that we will never cut at least $m/2$ edges is at most

$$\Big(1 - \frac{1}{2m}\Big)^{2000m} \leq 2^{-1000}$$

This can be generalized in the following lemma.

Lemma 12.2 ()

There is an algorithm that on input graph $G = (V, E)$ and a number k, runs in polynomial time in $|V|$ and k and outputs a cut $(S\overline{S})$ such that

$$\mathbb{P}\Big(\text{number of edges cut by } (S, \overline{S}) \geq \frac{|E|}{2}\Big) \geq 1 - 2^{-k}$$

Proof.

Just repeat the previous algorithm $200km$ times and compute the probability of failure.

12.1.3 Two-sided Amplification

The analysis above relied on the fact that the maximum has *one sided error*; that is, if we get a cut of size at least $m/2$ then we know we have succeeded. This is common for randomized algorithms, but it is not the only case. In particular, consider the task of computing some Boolean function $F : \{0,1\}^* \longrightarrow \{0,1\}$. A randomized algorithm A for computing F, given input x, might toss coins and succeed in outputting $F(x)$ with probability, say 0.9. We say that A has *two sided errors* if there is a positive probability that $A(x)$

outputs 1 when $F(x) = 0$ and positive probability that $A(x)$ outputs 0 when $F(x) = 1$. So, we cannot simply repeat it k times and output 1 if a single one of those repetitions resulted in 1, nor can we output 0 if a single one of the repetitions resulted in 0. But we can output the *majority value* of these repetitions: the probability that the fraction of the repetitions where A will output $F(x)$ will be at least, say 0.89, will be exceptionally close to 1 and in such cases we will output the correct answer.

Theorem 12.2 ()

If $F : \{0,1\}^* \longrightarrow \{0,1\}$ is a function such that there is a polynomial-time algorithm A satisfying

$$\mathbb{P}\big(A(x) = F(x)\big) \geq 0.51$$

for every $x \in \{0,1\}^*$, then there is a polynomial time algorithm B satisfying

$$\mathbb{P}\big(B(x) = F(x)\big) \geq 1 - 2^{-|x|}$$

for every $x \in \{0,1\}^*$.

12.1.4 Solving SAT through Randomization

The 3SAT problem is **NP** hard, and so it is unlikely that it has a polynomial (or even subexponential) time algorithm. But this does not mean that we can't do at least somewhat better than the trivial 2^n algorithm for n-variable 3SAT. The best known worst-case algorithms are randomized and are at their base the following simple algorithm. In this algorithm, called *WalkSAT*, the input is an n variable 3CNF formula φ, the parameters are any numbers $T, S \in \mathbb{N}$, and the operation is:

1. Repeat the following T steps:
   (a) Choose a random assignment $x \in \{0,1\}^n$ and repeat the following for S steps:
      i. If x satisfies φ, then output x.
      ii. Otherwise, choose a random clause $(l_i \vee l_j \vee l_k)$ that x does not satisfy, choose a random literal in $l_i \vee l_j \vee l_k$ and modify x to satisfy this literal.
2. If all the $T \cdot S$ repetitions above did not result in a satisfying assignment, then output `Unsatisfiable`.

Note that we are only going though at most $S \cdot T$ configurations of $x \in \{0,1\}^n$, and the running time of this algorithm is $S \cdot T \cdot poly(n)$. The fact that this algorithm is efficient is taken care of, so now the key question is how small we can make S and T so that the probability that WalkSAT outputs `Unsatisfiable` on a satisfiable formula φ is small. It is known that we can do with

$$ST = \tilde{O}\big((4/3)^n\big) = \tilde{O}(1.\overline{3}^n)$$

However, we will prove a weaker bound in the following theorem (which is still much better than the 2^n bound).

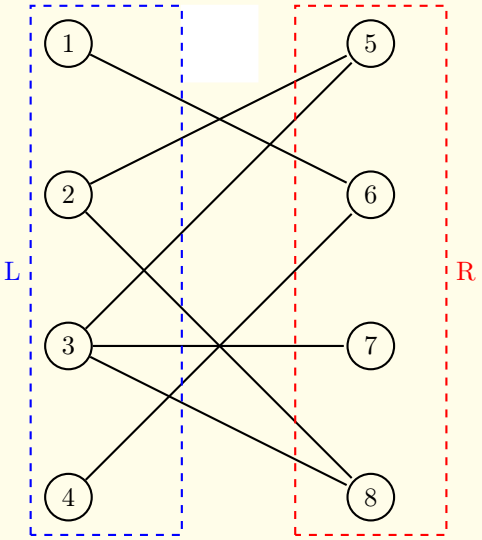
Theorem 12.3 (WalkSAT simple analysis)

If we set $T = 100\sqrt{3}^n$ and $S = n/2$, then the probability we output `Unsatisfiable` for a satisfiable φ is at most $\frac{1}{2}$.

12.1.5 Bipartite Matching

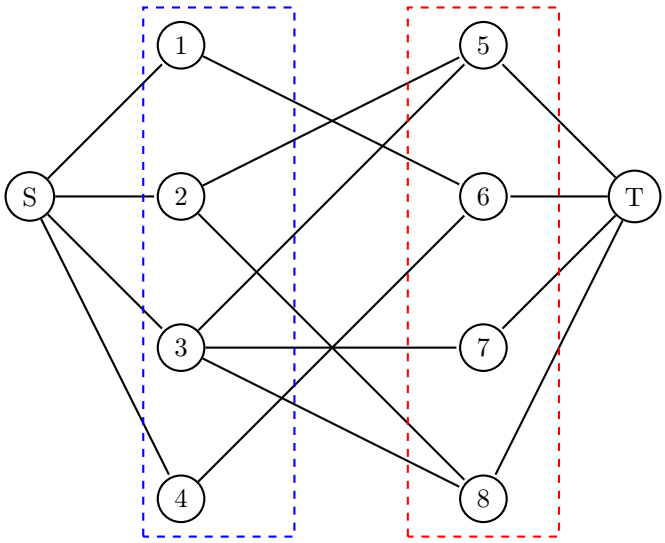
Definition 12.1 ()

A **bipartite graph** $G = (L \cup R, E)$ has $2n$ vertices partitioned into n-sized sets L and R, where all edges have one endpoint in L and the other in R.



A *matching problem* is a type of problem where we match nodes to each other with edges. One variant of it is called the *bipartite perfect matching*. The goal is to determine whether there is a *perfect matching*, a subset $M \subseteq E$ of n disjoint edges that connects every vertex L to a unique vertex in R.

It turns out that by reducing this problem of finding a matching in G to finding a maximum flow (or equivalently, a minimum s, t cut) in a related graph G' (below), we can solve it in polynomial time.



However, there is a different probabilistic algorithm to do this. Let G's vertices be labeled as $L = \{l_0, ..., l_{n-1}\}$ and $R = \{r_0, ..., r_{n-1}\}$. A matching M corresponds to a *permutation* $\pi \in S_n$ where for ever $i \in [n]$, we define $\pi(i)$ to be the unique j such that M contains the edge $\{l_i, r_j\}$. Define an $n \times n$ matrix $A = A(G)$ where $A_{i,j} = 1$ if and only if $\{l_i, r_j\}$ is present and $A_{i,j} = 0$ otherwise. The correspondence between matchings and permutations implies the following claim.

Lemma 12.3 (Matching polynomial)

Define $P = P(G)$ to be the polynomial mapping $\mathbb{R}^{n^2}$ to $\mathbb{R}$ where

$$P(x_{0,0}, ..., x_{n-1,n-1} = \sum_{\pi \in S_n} \left(\prod_{i=0}^{n-1} sign(\pi) A_{i,\pi(i)} \right) \prod_{i=0}^{n-1} x_{i,\pi(i)}$$

In fact, given the matrix A representing the graph, the polynomial above is the determinant of the matrix $A(x)$, which is obtained by replaying $A_{i,j}$ with $A_{i,j}x_{i,j}$. Then G has a perfect matching if and only if P is not identically zero (i.e. if there exists some assignment $x = (x_{i,j})_{i,j \in [n]} \in \mathbb{R}^{n^2}$ such that $P(x) \neq 0$.

This reduces testing perfect matching to testing whether a given polynomial $P(\cdot)$ is identically 0 or not. The kernel of most multivariate nonzero polynomials form a strictly lower dimensional space than the total space, so in order to do this, we just choose a "random" input x and check if $P(x) \neq 0$. However, to transform this into an actual algorithm, we can't work in the real numbers with our finite computational power. We use the following.

Theorem 12.4 (Schwartz-Zippel Lemma)

For every integer q and polynomial $P : \mathbb{R}^n \longrightarrow \mathbb{R}$ with integer coefficients, if P has degree at most d and is not identically zero, then it has at most dq^{n-1} roots in the set

$$[q]^n = \{(x_0, ..., x_{n-1}) \mid x_i \in \{0, 1, ..., q-1\}\}$$

Therefore, upon an input of a bipartite graph G on $2n$ vertices $\{l_0, ..., l_{n-1}, r_0, ..., r_{n-1}\}$, the *Perfect-Matching algorithm* can be divided into these steps:

1. For every $i, j \in [n]$, choose $x_{i,j}$ independently at random from $[2n] = \{0, ..., 2n-1\}$.
2. Compute the determinant of the matrix $A(x)$ whose i, jth entry equals $x_{i,j}$ if the edge $\{l_i, r_j\}$ is present and 0 otherwise.
3. Output `no perfect matching` if determinant is 0, and output `perfect matching` otherwise.

13 Modeling Randomized Computation

While we have described randomized algorithms in an informal way, we haven't addressed two questions:

1. How do we actually efficiently obtain random strings in the physical world?
2. What is the mathematical model for randomized computations, and is it more powerful than deterministic computation?

The first question is important, but we will assume that there are various physical sources of random or unpredictable data, such as a user's mouse movements, network latency, thermal noise, and radioactive decay. For example, many Intel chips come with a random number generator built in. We will focus on the second question.

13.1 Modeling Randomized Computation

Modeling randomized computation is actually quite easy. We can add the operation

```
foo = RAND()
```

in addition to things like the NAND operator to any programming language such as NAND-TM, NAND-RAM, NAND-CIRC, etc., where `foo` is assigned to a random bit in $\{0,1\}$ independently every time it is called. These are called RNAND-TM, RNAND-RAM, and RNAND-CIRC, respectively.

Similarly, we can easily define randomized Turing machines as Turing machines in which the transition function δ gets an extra input (in addition to the current state and symbol read from the tape) a bit b that in each step is chosen at random in $\{0,1\}$. Of course the function can ignore this bit (and have the same output regardless of whether $b = 0$ or $b = 1$) and hence randomized Turing machines generalize deterministic Turing machines.

We can use the `RAND()` operation to define the notion of a function being computed by a randomized $T(n)$ time algorithm for every nice time bound $T : \mathbb{N} \longrightarrow \mathbb{N}$, but we will only define the class of functions that are computable by randomized algorithms running in *polynomial time*.

Definition 13.1 (The class BPP)

Let $F : \{0,1\}^* \longrightarrow \{0,1\}$. We say that $F \in \mathbf{BPP}$ if there exist constants $a, b \in \mathbb{N}$ and a RNAND-TM program P such that for every $x \in \{0,1\}^*$, on input x, the program P halts within at most $a|x|^b$ steps and

$$\mathbb{P}\big(P(x) = F(x)\big) \geq \frac{2}{3}$$

where this probabilty is taken over the result of the RAND operations of P. Note that this probability is taken only over the random choices in the execution of P and *not* over the choice of the input x. That is, **BPP** is still a *worst case* complexity class, in the sense that if F is in **BPP** then there is a polynomial-time randomized algorithm that computes F with probability at least $2/3$ on *every possible* (and not just random) input.

We will use the name *polynomial time randomized algorithm* to denote a computation that can be modeled by a polynomial-time RNAND-TM program, RNAND-RAM program, or a randomized Turing machine.

Alternatively, we can think of a randomized algorithm A as a *deterministic algorithm* A' that takes two inputs x and r where the input r is chosen at random from $\{0,1\}^m$ for some $m \in \mathbb{N}$. The equivalence to the previous definition is shown in the following theorem:

Definition 13.2 (Alternative characterization of BPP)

Let $F : \{0,1\}^* \longrightarrow \{0,1\}$. Then $F \in \mathbf{BPP}$ if and only if there exists $a, b \in \mathbb{N}$ and $G : \{0,1\}^* \longrightarrow \{0,1\}$ such that G is in $\mathbf{P}$ and for every $x \in \{0,1\}^*$,

$$\mathbb{P}\big(G(xr) = F(x)\big) \geq \frac{2}{3}$$

where r is chosen at random from $\{0,1\}^{a|x|^b}$. As such, if A is a randomized algorithm that on inputs of length n makes at most m coin tosses, we will often use the notation $A(x;r)$ (where $x \in \{0,1\}^n$ and $r \in \{0,1\}^m$ to refer to the result of executing x when the coin tosses of A correspond to the coordinates of r. This second input r is sometimes called a **random tape**.

The relationship between **BPP** and **NP** is not known, but we do know the following.

Theorem 13.1 (Sipser-Gacs Theorem)

If $\mathbf{P} = \mathbf{NP}$ then $\mathbf{BPP} = \mathbf{P}$.

13.1.1 Success Amplification of two-sided error algorithms

The number 2/3 may seem arbitrary, but it can be amplified to our liking.

Theorem 13.2 (Amplification)

Let $F : \{0,1\}^* \longrightarrow \{0,1\}$ be a Boolean function such that there is a polynomial $p : \mathbb{N} \longrightarrow \mathbb{N}$ and a polynomial-time randomized algorithm A satisfying that for every $x \in \{0,1\}^n$,

$$\mathbb{P}\big(A(x) = F(x)\big) \geq \frac{1}{2} + \frac{1}{p(n)}$$

Then for every polynomial $q : \mathbb{N} \longrightarrow \mathbb{N}$, there is a polynomial-time randomized algorithm B satisfying for every $x \in \{0,1\}^n$,

$$\mathbb{P}\big(B(x) = F(x)\big) \geq 1 - 2^{-q(n)}$$

13.1.2 BPP and NP Completeness

The theory of **NP** completeness still applies to probabilistic algorithms.

Theorem 13.3 ()

Suppose that F is **NP** hard and $F \in \mathbf{BPP}$. Then

$$\mathbf{NP} \subseteq \mathbf{BPP}$$

That is, if there was a randomized polynomial time algorithm for any **NP** complete problem such as 3SAT, ISET, etc., then there would be such an algorithm for *every* problem in **NP**.

13.2 The Power of Randomization

To find out whether randomization can add power to computation (does **BPP**=**P**?), we prove a few statements about the relationship of **BPP** with other complexity classes.

Theorem 13.4 (Simulating randomized algorithms in exponential time)

$$\mathbf{BPP} \subseteq \mathbf{EXP}$$

Proof.

We can just enumerate over all the (exponentially many) choices for the random coins.

Furthermore,

$$\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{EXP}$$