Harvard University

Computation

Author:
Justin Zhu

February 1, 2020

Table of Contents

Chapter 1

Introduction

This book is adapted from Boaz Barak's *Introduction to Theoretical Computer Science* course taught at Harvard University. Many thanks to Professor Barak and the teaching staff for their hard work and support.

The first 19 chapters are dedicated to covering the big ideas of computation. The final chapters cover special topics in encryption and cryptography.

We converged on encryption and cryptography because we believe they are understated topics in a scientific era increasingly defined by data and computation. While mass and energy stand at the nexus of scientific advancements in the 20th century, data and code stand at the nexus of scientific advancements in the 21st century.

Better encryption practices start with education. In this era of big data and computation, understanding our data and code is just as important as understanding our constitutional rights and civil liberties, and this book seeks to make accessible the knowledge needed to better civic life for all in the 21st century.

Chapter 2

Math

2.1 Sets

Relations

$\in$ Membership

$\subseteq$ Containment

Operations

$\cup$ Union

$\cap$ Intersection

Set

$\times$ Cartesian product

2.2 Strings and Tuples

$\sum^k$ is lenght-k string over elements in $\sum$. If $\sum = 0, 1$, then

$$\{0,1\}^{10}$$

is the set of all 10 characters consisting 0 and 1. We denote $\sum^*$ as the strings of finite length.

[n] is the set $\{0, 1, 2 \cdots, n-1\}$ while $\mathbb{N} = \{0, 1, 2, \cdots\}$.

2.3 Functions

A function $f : S \to T$ is one-to-one (injective) if $\forall x, x' \in S$:

$$x \neq x' \implies f(x) \neq f(x')$$

2.4 Theorem

Refers to a significant result we want to remember and highlight

2.5 Lemma

Refers to technical result not necessarily important in its own right but useful in proving other theorems

2.6 Claim

A "Throw away" statement we use to prove bigger result and so we don't care much about it

2.7 Set Theory

Cardinality of finite set S is $|S|$, the number of elements it contains, and all sets contain the empty set $\emptyset$

The set difference is $S \backslash T$ all sets in S but not in T.

We can identify a sequence $(a_0, a_1, a_2, \ldots)$ of elements in some set S with a function $A : \mathbb{N} \to S$ (where $a_n = A(n)$ for every $n \in \mathbb{N}$).

Similarly, we can identify a k-tuple $(a_0, \ldots, a_{k-1})$ of elements in S with a function $A : [k] \to S$.)

2.8 Cartesian Product

If S, T, U are sets then $S \times T \times U$ is the set of all ordered tripes (s, t, u) where $s \in S, t \in T, u \in U$, generally, for every positive integer n and sets $S_0, \ldots, S_{n-1}$, we denote by $S_0 \times S_1 \times \cdots \times S_{n-1}$ the set of ordered n-tuples $(s_0, \ldots, s_{n-1})$ where $s_i \in S_i$ for every $i \in \{0, \ldots, n-1\}$. For every set S , we denote the set $S \times S$ by $S^2, S \times S \times S$ by $S^3, S \times S \times S \times S$ by S^4, and so on and so forth.

2.9 Special Sets

$$\{0,1\}^n = \{(x_0, \ldots, x_{n-1}) : x_0, \ldots, x_{n-1} \in \{0,1\}\}$$

is the set of all n-length binary strings for some natural number n, or the set of all n-tuples of zeroes or ones. This is consistent with our notation.

$$\{0,1\}^*$$

includes "string of length 0" or "the empty string," which includes ϵ or λ.

For every set $\sum$, we define $\sum^* = \cup_{n \in \mathbb{N}} \sum^n$

Concatenation is the addition of different strings $x \in \sum^n$ and $y \in \sum^m$ where $(n+m)$-length string xy of y after x.

The image of function F is the set $\{F(x)|x \in S\}$, the subset of F's codomain of all output elements mapped to the input. Image is equal to cardinality if injective.

We want to emphasize that a function f from A to B might not be total, so $f : A \to_p B$. The partial function F from S to T is also a total function S to $T \cup \{\perp\}$, and $\perp$ is a failure symbol.

2.10 Path

$$(u_0, \cdots, u_k) \in V^{k+1}$$

for $k > 0$ such that u_{i+i} is a beuggvir if u_i for every $i \in [k]$. The simple path is where they are distinct and they cycle is where $u_0 = u_k$. Two vertices $u, v \in V$ are connected if $u = v$ and there exists a path from $(u_0, \cdots, u_k)$ where $u_0 = u$ and $u_k = v$.

We denote by $\forall_{x \in S} P(x)$ the statement that is true if and only if $P(x)$ is true for every $x \in S$. We denote by $\exists x \in SP(x)$ the statement that is true only if there exists some $x \in S$ jsuch that $P(x)$ is true.

2.11 Free vs Bound Variables

A, B, C are Boolean circuits, S is Set, G is a graph, P, Q are programs, T is a function mapping the natural number to natural numbers with a time bound, and $\sum$ is a finite set. F, G, H are infinite functions mapping the variables for some m.

Chapter 3

Representations

A representation scheme maps an object to a binary string.

Recall that binary strings are represented

$$\{0,1\}^*$$

.

The representation (encoding) must be *one-to-one*.

3.1 Representing Natural Numbers

```
def NtS(n):
    if n > 1:
        return NtS(n // 2) + str(n % 2)
    else:
        return str(n % 2)
```

3.2 Representing Strings

```
def StN(x):
    return sum(int(x[i]) * 2 ** (len(x) - i - 1)) for i in range(len(x))
```

3.3 Representing Integers

Integers include the negatives of the natural numbers. We have two's complement representation of an integer seen as

$$\{-2^n, -2^n + 1, \cdots, 2^n - 1\}$$

The leading bit will denote the sign such that if $-2^n \leq k \leq -1$, then we would be feeding in $NtS_{n+1}(2^{n+1} + k)$.

Positive values are treated the same as the case for natural numbers.

3.4 Representing Rational Numbers

We represent using a pair of strings such that 10||110001 contains the numerator in the left and the denominator in the right.

3.5 Representing Real Numbers

There is no way! There are uncountably infinite real numbers and countable bits.

The mathematical justification is that there is no *one-to-one* mapping. This explains why we have floating point errors.

3.6 Representation Scheme

A representation scheme consists of:

1. An encoding function E
2. A decoding function D

$D(E(o)) = o$ for every $o \in O$.

3.7 Prefix-free Encoding

If our representation has the property that no string x representing an object o is a prefix of a string y representing a different object o'.

What are some common prefix-free encodings?

1. Every *fixed output length* is automatically prefix-free since a string x s a prefix of equal-length x' if x and x' are identical.
2. Map 0 to 00, 1 to 11, and an ending value to 01.

The proof is to break it down into the cases for two strings x and x':

1. If $|x| = |x'|$, then there is a coordinate in which they differ
2. If $|x| < |x'|$, $PF(x)$ will have the value 01 at some location, at which x' must have a value of either 00 or 11.

Chapter 4

Circuits

Circuits are programs! Circuits are defined by its gates.

4.1 Boolean Circuits

It consists of

- OR, AND, and NOT gates
- Wires representing 0 or 1.
- Inputs have outgoing wires

Boolean circuits are also represented as a Directed Acyclic Graph (DAG):

1. n inputs are vertices with no in-neighbors and at least one out-neighbor.
2. m outputs are vertices with no out-neighbors and at least one in-neighbor.
3. s gates are vertices with some number of in-neighbors and one out-neighbor.
4. Edges are wires.

5. Inputs are vertices with no incoming edges.
6. Size of Boolean circuit is number of gates contained

4.2 AON-CIRC

It is a straight-line programming language that describes Boolean circuits. The main usage is to demonstrate composition of AND, OR, and NOT.

4.3 NAND-CIRC

NAND-CIRC is a straight-line program that only uses the NAND function.

4.4 Circuit-Program Duality

For every f, that is computable by AON-CIRC and NAND-CIRC straight-line program of s lines, f is computable by a boolean and NAND circuit of s gates.

4.5 Straight-Line Programs

A $\mathcal{F}$ program is a sequence of lines assigning the variable into k_i other variables using input $X[i]$ and $Y[i]$ to denote input and output variables.

4.6 Transistors

A transistor contains an electric circuit with two inputs and one output (sink):

1. Source

2. Gate - controls whether current flows from the source to the sink.
3. Sink

Standard transistor: if the gate is ON, the current flows from the source to the sink. Complementary transistor behaves the opposite. The transistors are activated by a high enough voltage.

These transistors would double every year under Moore's law.

Chapter 5

Syntactical Sugar

Syntactical sugar is sophisticated features added to basic building blocks.

Every finite function can be computed by a Boolean circuit.

When solving problems, once we show that a computational model X is equivalent in power to the model and an additional feature Y, we can just use that model to show another function f as computable by X.

5.1 NAND-PROC-CIRC

NAND-PROC-CIRC is NAND-CIRC with syntactical sugar. We have the additional features:

- If statements
- Addition and Multiplication
- Lookup function $\leq 4 \cdot 2^k$ lines in NAND-CIRC

The code for computing XOR_S is repetitive, does not capturing the fact that there is a single algorithm to compute the parity on all inputs.

5.2 Definition of Algorithm

An algorithm is a "finite answer to an infinite number of questions." To give more expressive power to algorithms, we add syntactical sugar.

Creating an algorithm comprises of the following:

1. Finit set of instructions
2. Local variables or finite state
3. Unbounded Working memory to store input and unbounded transition variables
4. Address the parts in working memory to read and write to
5. Loops and Halts: we want to repeat instructions

5.3 Turing Machines and Circuits

Turing Machines add syntatical sugar onto circuits. There is the possibility of loops in Turing machines, and with it the possibility of infinite loops in Turing Machines.

5.4 Syntactical Sugar for infinite computation

Finite computation has circuits and straightline programs. Infinite computation has algorithms, Turing Machines, and programs.

To make infinite computation easier, we add:

1. Inner loops with while and for operations
2. Multiple index variables

3. Arrays with more than one dimension

4. GOTO statements that jump to certain lines in execution (not the best coding practice because it's harder to reason about invariants, according to Dijkstra)

Chapter 6

Code-Data

The power of computing is the code-data duality.

Code can evaluate other code, this is known as a "meta-circular evaluator."

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

This is because each variable can be encoded in $\log s$ length. We then encode the list of s variables so that we have $s \log s$.

6.1 EVAL

Signature:

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

Input:

1. String p
2. string x

Specification:

1. If p is a string that represents a list of tuples L such that (n, m, L) is a representation of $NAND - CIRC$ program P. Then $EVAL(px) = P(x)$

2. $EVAL_{s,n,m}$ is a finite function taking a string of fixed length as input and outputting a string of fixed length as output.

3. $EVAL_{s,n,m}$ is a single function, such that computing $EVAL_{s,n,m}$ allows to evaluate arbitrary NAND-CIRC programs of a certain length on arbitrary inputs of the appropriate length.

4. $EVAL$ is a function not a program.

$EVAL$ can be thought of an interpreter, and interpreters that are created using the same language it interprets demonstrates *self-circularity*.

6.2 Bounded Universal Program

The bounded universal program is the program that computes $EVAL_{s,n,m}$.

- "Universal" stands for a single program evalutating arbitrary code
- "Bounded" stands for $U_{s,n,m}$ evaluating programs of finite bounded size

A function of s lines takes at most $2s$ lines. Thus every finite function can be computed by some $NAND - CIRC$ program.

Bounding the Bounded Universal Program

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

6.3 Cake Theorem

We can translate every line of a Python program for $EVAL$ into an equivalent NAND-CIRC snippet.

We can generalize and say it is possible to translate every machine language program into an equivalent NAND-CIRC program of comparable efficiency.

It's important to understand how programs in high level languages such as Python are transformed into concrete low-level representation such as NAND.

6.4 Church-Turing Thesis

Any finite function can be computed by a Boolean circuit.

Chapter 7

NAND

NAND is universal because you can express any Boolean function using only NAND statements.

A $NAND - CIRC$ is then a circuit with only NAND gates.

7.1 Universality of NAND

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

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

There exists a NAND-CIRC program with at most $cm2^n$ lines computing the funcion f.

Every finite function can be computed by a large enough Boolean NAND circuit. How?

1. Define three lines for variables zero and one initialized to 0 and 1 respectively
2. Replacing a statement $Gxxx = 0$ with $Gxxx = NAND(one, one)$.
3. Replacing a statement $Gxxx = 1$ with $Gxxx = NAND(zero, zero)$.

4. $LOOKUP$ will be conducted using the $LOOKUP_n$ function created out of $NAND$ in our $NAND - CIRC$.

Runtime Analysis: $3 + 2^n$ lines for initializing the variables. $4\,(2^n)$ lines for computing $LOOKUP_n$.

The key idea is that we can simply represent a function as matching a set of inputs with a set of outputs.

7.2 Bounding the NAND-CIRC program

Thus, every function can be computed by a NAND-CIRC program of at most $O(m2^n/n)$ lines.

There is some constant $c > 0$ such that for every $n, m > 0$ and function f, there is a Boolean circuit with at most $cm2^n/n$ gates computing function f.

Proof

1. There are at most m output gates
2. There exists 2^{2^k} distinct functions mapping
$$\{0,1\}^k \to \{0,1\}$$
3. $k = \log(n - 2\log n)$
4. Define a function g:
$$g(a) = f\left(a0^{n-k}\right) f\left(a0^{n-k-1}1\right) \cdots f\left(a1^{n-k}\right)$$
5. Compute $f(x)$ by computing string $T = g(a)$ of length 2^{n-k}, and the computing $LOOKUP_{n-k}(T, b)$ to get the element of T corresponding to b.
6. The cost is then $cost(g) + O(2^{n-k})$

7.

$$O\left(2^{2^k}\cdot 2^k + 2^{n-k}\right) = O\left(2^{n-2\log n}\cdot(n-2\log n) + 2^{n-\log(n-2\log n)}\right) \leq$$
$$O\left(\frac{2^n}{n^2}\cdot n + \frac{2^n}{n-2\log n}\right) \leq O\left(\frac{2^n}{n} + \frac{2^n}{0.5n}\right) = O\left(\frac{2^n}{n}\right)$$

This is simpler because it avoids using all the syntactical sugar.

For circuits we have

1. f uses a Boolean circuit of at most $2n$ gates for each of the N functions δ_x, and combine that with at most N OR gates obtains a circuit of at most $2n \cdot N + N$ gates.
2. N is at most 2^n

7.3 SIZE(T)

$SIZE(s)$ is the set of functions that can be computed by NAND circuits of at most s gates.

1. $SIZE_{n,1}(s)$ is the the set of functions with input n that can be computed by NAND circuits of at most s gates.
2. $SIZE(s) = \bigcup_{n,m\leq 2s} SIZE_{n,m}(s)$

We say that $f \in SIZE_{n,1}(s)$ if the smallest circuit that computes f has s or fewer gates.

$SIZE_{n,1}\left(c \cdot 2^n/n\right)$ corresponds to the set of all func- tions from $\{0,1\}^n$ to $\{0,1\}$ since we proved that every function can be computed by a NAND-CIRC program of at most $O(m2^n/n)$ lines.

7.4 Size Comparison of NAND-CIRC and AON-CIRC

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

Chapter 8

Turing Machine

8.1 Turing machines

Turing machines are invented by Alan Turing in 1936, describing finite algorithm handling arbitrary long inputs.

8.2 Turing equivalence

Turing equivalence, or Turing completeness, describes the many computational models (Python, C, Lisp) that are fundamentally equivalent to a Turing machine.

8.3 Innards of a Turing Machine

1. Machine is at state 0
2. Uninitialized tape
3. Location i t which the machine points is set to 0
4. The machine reads the symbole $\sigma = T[i]$ that is in the i-th location of the tape

5. The rules the Turing machine follows is the transition function δ_M with $k|\Sigma|$ inputs and $4k|\Sigma|$ outputs.

6. Halting is obtained by reading the tape from the second location, the output is concetenating all the symbols in $T[0], \cdots, T[i]$. where i is the final head position.

8.4 Inputs to Turing Machine

1. Tape with input $x \in \{0,1\}^*$

2. Function F

8.5 PAL Example

Specification: PAL is a function that on input $x \in \{0,1\}$ outputs 1 if and only if

1. x is an even-length palindrome

2. Mathematically, x is expressed as $w_0 \cdots w_{n-1} w_{n-1} \cdots w_0$ for $w \in \{0,1\}^*$

The Turing machine M that computes PAL is comprised of the following

1. M's tape alphabet $\{0, 1, stop, start, empty - cell\}$

2. M's transition function containing $k = 14$

3. The algorithm (list of steps) for computing PAl using the tape alphabet and transition function

8.6 Turing Machines and Python

Effectively, both Turing Machines and Python are programs. The only difference is the syntax:

1. Turing Machine Tape becomes Python list or array holding values from finite set Σ
2. Turing Machine Head position becomes Python integer variable of unbounded size
3. Turing Machine State becomes Python variable of fixed size

```
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 transitio
        if Tape[i]=="0" and state==7:
            i += 1
            Tape[i]="1"
            state = 19
        elif Tape[i]==">" and state == 13:
            Tape[i]="0"
            state = 15
        elif ...
        ...
        elif Tape[i]==">" and state == 29:
            break # Halt
```

Chapter 9

NAND-TM

NAND-TM models a single uniform algorithm that takes inputs of *arbitrary* lengths. This is achieved using loops and arrays.

9.1 Loops

NAND-CIRC is a straight-linie programming language which means that a NAND-CIRC program of s lines takes s steps of computation.

NAND-TM takes arbitrary steps of computation.

9.2 Arrays

NAND-CIRC program of s lines stores a maximum of $3s$ variables.

NAND-TM takes stores arbitrary number of variables.

9.3 NAND-TM and NAND-CIRC

NAND-TM = NAND-CIRC + loops + arrays

9.4 NAND-TM Syntax

1. Special integer valued variable i which is Boolean valued
2. Scalar variables hold one bit and are lowercase
3. Array variables hold unbounded number of bits and are uppercase
4. MODANDJUMP(a,b) which decrements/increments i and j, in addition to deciding whether to jump to the first line of the program, continues, or halts. This appears in the last line of the NAND-TM program
5. All values are 0 or 1

9.5 NAND-TM and Turing Machines

Turng	NAND-TM
Tape: finite length	Array: unbounded length
Head location	Index Variable
Accessing Memory: can only access tape at current head location	Accessing Memory: Can access all the scalar variables, but only location of the index variable i in the array
Machine moves head by at most one	Main loop moves index by at most 1

Turing machines and NAND-TM are equivalent because a function computable by NAND-TM program P is computable by Turing Machine M and vice versa.

Chapter 10

Function-Programs

Functions are not programs!

Function is the specification (what to compute) while Program (algorithms/circuits/machine) is the implementation (how to compute).

To beat the horse, a program computes a function if for every input x, we follow the instructions of program P on the input x to obtain the output $F(x)$.

10.1 Computational Process

A *computational process* takes inputs containing a string of bits and outputs a string of bits. It has:

- Specification (Function)
- Implementation (Program)

10.2 Computable Functions

Let F be a total function and M be a Turing machine that computes it.

The very existence of a Turing machine to compute F makes F a computable function.

10.3 R

In this book, R is the set of all computable functions. This is not to be confused with the mathematical R or the programming language R.

10.4 Language and Functions

"Language" is a set of values such that $F(x) = 1$ if x is a part of the language, explicitely, $x \in L$ where L is the langauge.

10.5 Partial Functions

Partial function are computble functions on inputs x that are defined.

Another way to phrase it: Turing Machine M halts on x if and only if F is defined on x.

Chapter 11

Uniformity

Uniformity of computation is where we can compute functions of all input lengths.

Turing Machines / NAND-TM are uniform because they have one program that computes for any input length, while NAND-CIRC is nonuniform because there needs to be a different program for different input lengths.

Uniformity enables the inputs and outputs to be longer than the code itself, implying "self-replication" properties.

11.1 Universality

Universality is defined as a single circuit that can evaluate all other circuits.

A universal Turing machine can evaluate all other machines, including ones that are more complex than itself.

11.2 Universal Turing machine

The universal Turing machine U computes $M(x) = y$, such that $U(M, x) = y = M(x)$. It evaluates arbitrary algorithms on arbitrary inputs, and is an interpreter.

11.3 Representation

We can represent Turing machines as strings. Given the string representation of Turing machine M and input x, we can simulate M's execution on input x.

11.4 Meta-circular Evaluator

Meta-circular evaluators are interpreters written in the language which they are supposed to interpret.

McCarthy's 1960 paper defined the Lisp language, where a Lisp function can evaluate any Lisp program, the beginning of the Meta-circular evaluator.

Chapter 12

Uncomputability

There exists a function that is not computable by any Turing machine.

The proof is in diagonalization: Write down a table for every possible input x and computable program y, and the cell corresponding to (x, y) would be $M_y(x)$.

$$F^*(x)$$

is the diagonal of this table such that $F(x)^* = 1 - M_y(x)$, and we notice it differs in one value with every single machine and it is not computable.

12.1 HALT

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

HALT is not computable.

The uncomputability of F^* implies uncomputability of HALT. Suppose there exists Turing machine M that can compute HALT, so we use that to obtain Turing Machine M' to compute F^*.

Let $x(x)$ be Turing machine described by string x on input x.

1. If $x(x) = 0$, then $HALT(x, x) = 1$. $M(x, x) = HALT(x, x)$ such that the value z will have value 1. Then $y = x(x) = 0$, outputting correct value of 1.

2. If the machine described by x does not halt on input x, then $HALT(x, x) = 0$. If M computes HALT, then machine M halts, outputting $z = 0$, outputting 0.

3. If the machine described by x halts on input x, it outputs some $y' \neq 0$. Then, $HALT(x, x) = 1$ and $y = y' \neq 0$, and so output 0.

$M'(x) = F^*(x)$, which implies that F^* is uncomputable, contradicting the assumption that M computes HALT.

12.2 Reductions

Essentially a contrapositive argument where if there exists a Turing machine that computes A, then there exists a Turing machine that computes $HALT$. We know $HALT$ is uncomputable so A therefore must not be computable.

$HALT(M, x) = A(R(M, x))$.

We reduce the task of computing $HALT$ to the task of computing A.

A reduction is an algorithm which means that it has three components:

1. Specification: $F(w) = G(R(w))$

2. Implementation: How to transform w to $R(w)$

3. Analysis: Proof for how $F(w) = G(R(w))$

12.3 HALTONZERO

is uncomputable.

12.4 ZEROFUNC

is uncomputable. $ZEROFUNC(M) = 1$ if and only if M represents a Turing machine such that M outputs 0 on every input, then $ZEROFUNC$ is uncomputable.

12.5 Semantic Property

Semantic properties mean properties of the function that the program computes, as opposed to properties that depend on particular syntax used by the program.

Thus, a semantic property is true for both programs or false for both programs, since it depends on the function the programs compute and not on their code.

A function is semantic if for every pair of strings M, M' that represent functionally equivalent Turing machines, $F(M) = F(M')$.

12.6 Rice's Theorem

If F is semantic and non-trivial then it is uncomputable.

Proof:

1. If F is nontrivial then there are two machines M_0 and M_1 such that $F(M_0) = 0$ and $F(M_1) = 1$.
2. If $F(P) = F'(P)$ for functionally equivalent programs P, P'

Chapter 13

Models

Models in computation include

1. Uniform computation models
2. Nonuniform computation models

As covered previously, Turing machines, NAND-TM, and standard programming languages are uniform computation models.

Chapter 14

Problems

We use running time as a way to express the efficiency of a function, which is represented as a problem. The running time of a problem is not a number, but rather a function of the length of the input.

There are two types of running-type algorithms: **polynomial** and **exponential**. These algorithms are *insensitive* to the computational model (Turing machines, λ-calculus, and Python are examples of computational models).

14.1 Graph Problems

A graph is comprised of

- V vertices, represented as $[n]$, the set of integers $\{0 \cdots 1\}$
- E edges (pairs of vertices), represented as (u, v)

There exists two characters of graphs

- Directed (source to sink) / Undirected (no sources or sinks)
- Simple (no parallel edges or self-loops) or general

Graphs, expressed mathematically as $G = (V, E)$ can be represented in adjacency list, traversing takes $O(n^2)$ computational runtime.

Graphs are useful:

1. Networks
2. Correlations in data
3. Causal relationships
4. State systems

14.2 Shortest Path Problem

1. Given graph $G = (V, E)$ and vertices $s, t \in V$
2. Find length of shortest path between s and t
3. Find $\min k$ subject to
$$v_0, \cdots, v_k$$
where $v_0 = s$ and $v_k = t$.

The optimal algorithm is to run BFS, computing in $O(n^2)$ time. Dijkstra's algorithm generalizes BFS to weighted graphs.

14.3 Longest Path Problem

This generalizes the Hamiltonian path problem, which asks for maximally long simple path, as well as the Traveling salesman problem visitng all vertices at most w.

The runtime for this is $O(c^n)$.

14.4 Minimum Cut Problem

A cut is defined as a subset $S \subseteq V$ such that S is not empty and not V.

1. The edges cut by S are edges with one endpoint in S and one endpoint in $V \setminus S$.
2. The set of edges where one enpoint is in S and the other is not in S:
$$E(S, \bar{S})$$
3. Find minimum $E(S, \bar{S})$ for $s \in S$ and $t \in V \setminus S$.

The naive algorithm for $MINCUT$ is 2^n, which is exponential.

$MINCUT$ could be reduced to *polynomial* in number of vertices.

14.5 Min-Cut Max-Flow

The minimum cut between s and t equals the maximum *flow* we can send from s to t if every edge has unit capacity.

The maximum s, t flow is always at most the value of the minimum s, t cut. This is because each cut in the minimum cut reduces an important flow in the maximum flow set.

14.6 Linear Programming

A flow on graph G of m edges can be modeled as a vector $x \in \mathbb{R}^m$.

1. For every edge $e = (u, v)$, x_e corresponds to water per time-unit flowing on e.
2. Water leaving source s is the same as water entering sink t:
$$\sum_{e \ni s} x_e + \sum_{e \ni t} x_e = 0$$

3. For every vertex v, the amount of water entering and leaving v is the same:
$$\sum_{e \ni v} x_e = 0 \quad \forall_{v \in V \setminus \{s,t\}}$$

4. Each edge has capacity one:
$$-1 \leq x_e \leq 1 \quad \forall_{e \in E}$$

   Summing over $e \ni v$ means summing all the edges that touch v.

The maximization function is
$$\sum_{e \ni s} x_e$$
over all vectors $x \in \mathbb{R}^m$. We can also express it as
$$\sum_{e \ni t} x_e$$
over all vectors $x \in \mathbb{R}^m$, since we know from our constraints that the flow out from s is equal to the flow into t.

The best runtime so far is
$$O\left(\min\left\{m^{10/7}, m\sqrt{n}\right\}\right)$$
, a polynomial algorithm.

14.7 Maximum Cut Problem

1. Maximum cut finds on an input graph $G = (V, E)$ the subset $S \subseteq V$ that maximizes edges cut by S.

Check out: Ising model and VLSI design

The runtime is $O(2^n)$, not polynomial.

Chapter 15

Reductions

NP problems have the following properties:

1. **Search** problems where we search for a solution that is good
2. Each of these problems have a trivial exponential time algorithm to enumerate all possible solutions
3. The best known algorithm is not faster than the trivial one in the worst case

If **NP** problem **A** reduces to **NP** problem **B**, we say that **B** is **NP**-complete.

15.1 Definition of Reduction

If **A** reduces to **B**, then if there exists a polynomial-time algorithm for **B** there must exist a polynomial-time algorithm for **A**.

The contrapositive is also true: If there does not exist a polynomial-time algorithm for **A**, then there does not exist a polynomial-time algorithm for **B**.

We can think of **A** reducing to **B** in the following python code.

```
def A(x):
  f = createReduction
  B(f(x))
```

Notice here that running algorithm **A** must mean running its subfunction algorithm corresponding to **B** after creating a reduction that runs polynomially.

Then if **B** runs polynomially, then **A** must surely run polynomially. However, if **A** does not run polynomially, then **B** couldn't possibly run polynomially, since we know *createReduction* runs polynomially.

The meat of a reduction problem is how we define our *createReduction* function.

15.2 3SAT

Representation:

$$3SAT : \{0,1\}^* \to \{0,1\}$$

.

Definition:

Given some

$$C_0 \wedge \cdots \wedge C_{m-1}$$

, where C_i is the OR of three variables and their negation (representation

$$C_i = x_0 \vee x_1 \vee x_2$$

), we return 1 if there exists an assignment of variables that cause the evaluation of

$$C_0 \wedge \cdots \wedge C_{m-1}$$

to TRUE.

15.3 Polynomial-time Reductions

Polynomial-time reductions are reductions with a special property:

B, the reduced function, is polynomial.

15.4 Formal Definition

If **A** reduces to **B**, then if there exists a polynomial-time algorithm for **B** there must exist a polynomial-time algorithm for **A**.

B is constructed to have a polynomial algorithm, and so the algorithm that solves for **A** is not harder than a polynomial-time algorithm, because **A** is no harder than **B**.

The contrapositive does not apply here: If there does not exist a polynomial-time algorithm for **A**, then there does not exist a polynomial-time algorithm for **B**. This is because we know that **B** is polynomial.

We represent this polynomial-time reduction mathematically as

$$A \leq_p B$$

15.5 Proving Reductions

To prove a reduction for $A \leq_p B$, follow a three-step process.

1. Create algorithm R that maps input for A into input for B.
2. Prove the algorithm runs in polynomial time
3. Prove $A(x) = B(R(x))$ for every x

15.6 Completeness

If $A(x) = 1$, then $B(R(x)) = 1$.

15.7 Soundness

If $B(R(x)) = 1$ then $A(x) = 1$. Alternatively, we can use the contropositive where $A(x) = 0$, then $B(R(x)) = 0$.

Chapter 16

NP

16.1 NP Definition

We define **NP** to be the class that contains all Boolean functions corresponding to a search problem. In simpler terms, F is in **NP** if there is a solution w of length polynomial that can be verified by polynomial-time algorithm V.

NP does not mean that the solution is not-P. Rather it means that verification of a solution is in polynomial time!

16.2 Proving NP

To show that something is in **NP**, we just need to define a certficate w a verifier V that can correctly compute in polynomial time, outputting 1 after verifying a correct solution to **NP** and 0 on all other inputs.

$P \subseteq NP \subseteq EXP$

If $F \in P$, then we can define our verifier to be in P, which means, $F \in NP$.

16.3 NP-Hard

G is **NP-Hard** if for every $F \in NP$, $F \leq_p G$. To show something is NP-hard we need to reduce one NP-hard F to G. This is because we know that F reduces to G and F is NP-hard, then G is at least NP-hard. To formally show this reduction of F to G, we need the following:

1. Describe the reduction algorithm
2. Show the reduction of F to G is polynomial with respect to the inputs (is in polynomial time)
3. Show completeness: If $F(x) = 1$, then $G(x) = 1$
4. Show soundness: If $G(x) = 1$, then $F(x) = 1$. Alternatively, we can use the contrapositive, which is if $F(x) = 0$, then $G(x) = 0$.

16.4 NP-Complete

G is **NP**-Complete if G is NP-hard and $G \in NP$.

16.5 Well-Known NP-Complete Problems

1. 3SAT
2. Independent Set
3. Maximum Cut
4. Subset sum
5. 01 Linear Equations
6. Integer Programming

16.6 Cook-Levin Theorem

For every $F \in NP, F \leq_p 3SAT$.

The Cook-Levin Theorem implies that an efficient algorithm in one of these NP problems will imply an efficient algorithm for all of the problems in NP.

This can be proved using the fact that for every $F \in \mathbf{NP}$:

$$F \leq_p NANDSAT \leq_p 3NAND \leq_p 3SAT$$

Chapter 17

P-NP

A major theoretical breakthrough would be if $P = NP$.

To show $P = NP$, then we would have to just give a polynomial-time algorithm for a single one of **NP**-complete problems.

To show $P \neq NP$, then we would have to show there doesn't exist a polynomial-time algorithm for every one of the **NP**-complete problems.

Neither has been shown to date.

17.1 Ladner's Theorem

If $P \neq NP$, there would exist problems that are neither in P nor are NP-complete.

17.2 3SAT

If $P = NP$, then $3SAT$ has an $O(n)$ or $O(n^2)$ algorithm.

If $P \neq NP$, then $3SAT$ cannot be solved faster than 2^{ϵ} for some tiny $\epsilon > 0$.

Chapter 18

Running Time

Running time is formally defines s measuring the number of steps as a function of the *length* of the input.

18.1 Turing Machine Definition

A function is computable in $T(n)$ if there exists a Turing Machine M such that for large n,

1. The machine M halts after executing $\leq T(n)$ steps
2. The machine outputs $F(x)$

18.2 Boolean Functions Definition

$TIME_{TM}(T(n))$ is the set of Boolean functions computable in $T(n)$ time.

18.3 RAM Machine Definition

$TIME_{RAM}(T(n))$ is the set of Boolean functions that are computable in $T(n)$ steps by a NAND-RAM program P.

$$TIME_{TM}(T(n)) \subseteq TIME_{RAM}(T(n)) \subseteq TIME_{TM}(T(n)^4)$$

18.4 Proving Time Bounds

How would we prove that $TIME_{TM}(10n^3) \subseteq TIME_{TM}(2^n)$?

1. Suppose $F \in TIME_{TM}(10n^3)$.
2. Set some large number N_0 such that for every $n > N_0$, $M(x)$ outputs $F(x)$ at most $10n^3$ steps.
3. $10n^3 = o(2^n)$
4. Thus, there exists $n > N_1$, such that $10n^3 < 2^n$
5. We observe
$$n > \max\{N_0, N_1\}.$$
$M(x)$ outputs $F(x)$ within 2^n steps and thus $F \in TIME_{TM}(2^n)$

18.5 Polynomial Time: P

A function is computable in polynomial time if:

$$F \in \bigcup_{c \in \{1,2,3\cdots\}} TIME(n^c)$$

Practically, this means there exists a Turing Machine that halts within at most $p(|x|)$ steps and outputs $F(x)$.

18.6 Exponential Time: EXP

A function is computable in polynomial time if:

$$F \in \bigcup_{c \in \{1,2,3\cdots\}} TIME(2^{n^c})$$

Note that c is a constant in both cases.

Practically, this means there exists a Turing Machine that halts within at most $2^{p(|x|)}$ steps and outputs $F(x)$.

Chapter 19

Time

Functions are classified by their running time. Here, we explore these different classes of functions according to their running time.

As defined in the last chapter, P is the class of functions where there exists a Turing Machine that halts within at most $p(|x|)$ steps and outputs $F(x)$.

EXP is the class of functions where there exists a Turing Machine that halts within at most $2^{p(|x|)}$ steps and outputs $F(x)$.

$$P \subseteq EXP$$

19.1 NAND-RAM

A function F is computable in $T(n)$ RAM time if there exists a NAND-RAM program P such that for every sufficiently large n and every $x = 0, 1^n$ input, the program P halts after executing $T(n)$ lines, outputting $F(x)$.

$TIME_{RAM}(T(n))$ is the set of functions computable in $T(n)$ RAM time. RAM is random access memory and serves as a paradigm for Turing machines in calculating $O(n)$ or $O(n \log n)$ time.

$$\text{TIMETM}(T(n)) \subseteq \text{TIMERAM}(10 \cdot T(n)) \subseteq \text{TIMETM}\left(T(n)^4\right)$$

All "reasonable" computational models are equivalent if we only care about the distinction between polynomial and exponential.

19.2 Relationships

P contains everything of the form $TIME(O(n^c))$ while EXP includes everything in P plus $TIME(n^{\log n})$, $TIME(2^n)$, and $TIME(2^{n^c})$. Note that EXP does not include $TIME(2^{2^n})$.

19.3 Church-Turing Thesis

The set of computable functions is the same for all physically realizable models.

Once again, this is because we can still treat programs as strings and have a universal program such that we still have time hierarchy and uncomputability results. There is no reason to doubt the ("plain") Church-Turing thesis.

19.4 Extended Church-Turing Thesis

The set of computable functions is the same for all physically realizable models, but the runtime difference between different models is at most polynomial with respect to the length of the inputs.

19.5 P/ poly

$$\mathbf{P}_{/\text{poly}} = \cup_{c \in \mathbb{N}} SIZE\left(n^c\right)$$

$F \in P$ means there is a single Turing machine computing F on all inputs in polynomial time.

$F \in \mathbf{P}/$ poly means that there for every input length n there can be a different circuit C_n that computes F using polynomially many gates on the inputs of these lengths.

19.6 Time-Size relationship

$$\text{TIME}(T(n)) \subseteq \text{SIZE}\,(T(n)^a)$$

Hence $\mathbf{P} \subseteq \mathbf{P}/$ poly

Chapter 20

Randomized Computation

20.1 Modeling Randomized Cmputation

To model randomized computation, we simply add the following operation to the programming language NAND-TM, CIRC, etc:

```
RAND()
```

To actually efficiently obtain random strings in the physical world, we can apply a randomness extractor[1] that transforms sources of randomness into a uniform random variable.

Randomized Turing Machines RNAND-TM are Turiing Machines where the transition function δ gets an extra input b which signifies a bit.

20.2 BPP

We define **BPP** to be the class that contains all functions where:

1. A RNAND-TM computes the function in at most $a|X|^b$ steps

[1] https://www.wikiwand.com/en/Randomness_extractor

2. $\Pr[P(x) = F(x)] \geq \frac{2}{3}$

BPP is a worst case complexity class such that if F is in **BPP**, there is a polynomial-time randomized algorithm that computes F with probability at least 2/3 on every possible input.

We can also think of this by saying if $F \in BPP$ if and only if there exists a G in **P** such that

$$\Pr_{r \sim \{0,1\}^{a|x|^b}}[G(xr) = F(x)] \geq \frac{2}{3}$$

We say r is a random tape that can take on any value of 0 or 1.

20.3 BPP vs NP

1. NP is one sided: $F(x) = 1$ if there exists a solution w such that $G(xw) = 1$ and $F(x) = 0$ if for every string w of the same length $G(xw) = 0$.
2. BPP is symmetric: $P(x) = F(x) = 0$ and $P(x) = F(x) = 1$ have the same probability

20.4 Amplification

If there exists a function

$$\Pr[A(x) = F(x)] \geq \frac{1}{2} + \frac{1}{p(n)},$$

then there must exist a function

$$\Pr[B(x) = F(x)] \geq 1 - 2^{-q(n)}.$$

We simply repeat running the experiment $O(k/\epsilon^2)$ times and apply Chernoff's inequality.

Chapter 21

Cryptography

Cryptography is the science of creating secrecy, a process known as encryption. It requires adding randomness, or noise, to information that needs to be transmitted.

Classical cryptography schemes have functions mapping natural numbers to natural numbers. This function mapping is known as an encryption scheme.

21.1 Encryption schemes

It is comprised of:

1. Plaintext length function $L : \mathbb{N} \to \mathbb{N}$
2. Ciphertext length function $C : \mathbb{N} \to \mathbb{N}$
3. Encryption and Decryption Functions (E, D)
4. $D(k, E(k, x)) = x$

Decoding and encoding yields the value x as desired. Then we have a codomain being at least as large as the domain, since the domain is $\{0,1\}^{L(n)}$ and codomain is $\{0,1\}^{C(n)}$ so $C(n) \geq L(n)$.

21.2 Randomness

The key to great encryption scheme is the randomness of the key not the obscurity of the encryption algorithm itself.

21.3 Perfect secrecy

An encryption scheme is secure if it is not possible to recover key k from $E_k(x)$.

In a stream cipher, we use a pseudorandom generator to obtain an encryption scheme with a key length of n and plaintext of length L. We can encrypt the plaintext with key k by ciphertext, and the plaintext must be smaller or equal to the length of the ciphertext.

We can verify perfect secrecy mathematically by the fact that the probability of any one encryption value being expressed is uniform across the sample space of all encryption values.

21.4 Semantic security

Semantic security is where negligible information about plaintext can be obtained from ciphertext (recall that in order for plaintext length must be smaller than ciphertext length). Then any polynomial probabilistic algorithm given the ciphertext of a certain message cannot determine any partial information on the message with probability higher than all other polynomial probabilistic algorithms that only have access to the message length and not the ciphertext

In summary, perfect secrecy means that ciphertext reveals no information about plaintext, whereas semantic security implies that any information revealed cannot be feasibly extracted. Both are equivalent statements, because we can prove an semantic security from perfect secrecy and we can prove perfect serecy from semantic security.

Chapter 22

One-way functions

A function

$$f : \{0,1\}^* \to \{0,1\}^*$$

is one-way if f can be computed by a polynomial time algorithm, but any polynomial time randomized algorithm F that attempts to compute a pseudo-inverse for f succeeds with negligible probability.

We can express this property explicitely for all randomized algorithms F, all positive integers c and all sufficiently large n, the length of x to be:

$$P(f(F(f(x))) = f(x)) < n^{-c}$$

F is a measure of randomness and P is the probability of choosing x is on the discrete uniform distribution on $\{0,1\}^*$.

The theoretical implications of proving the existence of one-way functions would be proving that there does not exist a natural proof for $P \neq NP$. This is because if f is a one-way function, then the inversion of f would be a problem whose output is hard to compute but easy to check, implying $P \neq NP$. The scope of this project will be dedicated to understanding how to verify if a function demonstrates properties of a one-way function, and identifying ways to prove the existence of one-way functions.

22.1 Examples

To prove whether a function is one-way, we have to find an efficient way of producing an efficient inverting algorithm for these functions.

Possible functions to investigate in finding an efficient inverting algorithm include the following:

Multiplication and Factoring

The function f takes as inputs two prime numbers p and q in binary notation and returns their product. Inverting this function requires finding the factors[1] of a given integer N, which is a one-way function.

Rabin function

The Rabin function squres modulo $N = pq$ for prime numbers p and q. Then

$$\text{Rabin}_N(x) \triangleq x^2 \bmod N$$

. Inverting the Rabin function would mean extracting square roots, which is a one-way function.

[1] https://www.wikiwand.com/en/Integer_factorization

Chapter 23

Computation Security

How would P = NP affect encryption security?

23.1 P-NP and One-Way Functions

A proof of P = NP would mean that one-way functions do not exist. Recall that we defined a one way function as f which can be computed by a polynomial time algorithm, but any polynomial time randomized algorithm F that attempts to compute a pseudo-inverse for f succeeds with negligible probability.

Recall that P = NP would imply that BPP is not in EXP. Therefore, if f could be computed by a polynomial time algorithm, then f could be computed by any polynomial time randomized algorithm F that attempts to compute a pseudo-inverse for f succeeding with negligible probability, since BPP is now not in EXP.

This necessarily implies that no provably secure encryption scheme can exist secrecy, since we would be able to iterate through all possible encryption schemes in polynomial time, rendering many security no longer "provably" secure.

23.2 Provable Security

Provable security refers to any type or level of security that can be proved. We can frame this in the following problem ISPROVABLESECURE:

Suppose an attacker must solve an NP-hard problem in order to break the security of the modelled system. If the NP-hard problem is solvable in polynomial time, then ISPROVABLESECURE is solvable in polynomial time. If ISPROVABLESECURE is not solvable in polynomial time, then the NP-hard problem is not solvable in polynomial time. In other words, ISPROVABLESECURE reduces to an NP-hard problem.

$$ISPROVABLESECURE \leq_p NP - Hard$$

Provable security is synonymous with secure coding and security by design, and both use proofs to show that the security.

23.3 Proof of Security

To show that a problem exhibits provable security, we would need to provide a proof of security consisting of the following two parts:

- A model of the system, defined as our ISPROVABLESECURE problem
- An attacker model M, representable as a Turing machine, that solves ISPROVABLESECURE would be uncomputable

23.4 Theory vs. Practice

Provable security is a theoretical property of security systems. In real life, encryption schemes are exposed to other vulnerabilities such as side-channel attacks (keyboard clicks, cache design, and memory layout, etc.) that are specific to the hardware and software housing said encryption scheme.

An encryption scheme that is provably secure does not mean that is incapable of being broken. Conversely, an encryption scheme that is not provably secure

does not mean that is capable of being broken in practice. This is because in practice, the implementation of the system often can add more security or less security to the encryption scheme. Moreover, the runtime in which encryption schemes can be broken also does make a difference. An algorithm that breaks an encryption scheme in $O(10000n)$ runtime almost is sometimes no different than an algorithm that breaks an encryption scheme in $O(2^n)$ runtime.