DUKE UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

Computer Science

Personal Notes

Muchang Bahng

August 7, 2021

Contents

Chapter 1

Introduction

This are my notes on computer science. Self studying starting from Jan 1st, 2021. We will assume sufficient background knowledge in mathematics, especially concerning set theory and basic facts about functions.

1.1 Prerequisites

Quick Notations

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

Definition 1.1.1. 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 1.1.2. The logarithm without any base will denote logarithm in base 2. That is,

$$\log n = \log_2 n$$

Definition 1.1.3. 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. 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 1.1.4 (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 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

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

1.2 Computers

A computer has many parts which are worth remembering.

Definition 1.2.1. The 5 main parts of a computer is:

1. The **Solid State Drive (SSD)** (or an older version is **HDD**) is where the computer's long term memory is stored. Most computers usually have 256GB, 512GB, or 1TB of storage.
2. The **Random Access Memory (RAM)** is where the computer's short term memory is stored, which is usually 4GB, 8GB, 16GB, or 32GB. The RAM determines

how well your computer can work with multiple things running at the same time (applications, browser tabs, ...). Accessing this memory is about 20 100 times faster than accessing the SSD memory.

3. The **processor**, or **Central Processing Unit (CPU)**, is the circuitry that executes instructions that make up a computer program. At the hardware level, a CPU is an **integrated circuit**, or a **chip**. This means that a CPU integrates billions of circuits into a chip, where each circuit is really just logic gate (AND, OR, NOT, etc.) made up of a transistor. We can see this in layers:

   $$\text{CPU Chip} \implies \text{Circuit} \implies \text{Logic Gate} \implies \text{Transistor}$$

   They're effectively minute gates that switch on or off, thereby conveying the ones or zeros that translate into everything you do with the device. One of the most common advancements of CPU technology is in making those transistors smaller and smaller, where the rate of improvement is referred to as *Moore's Law.*

   In simplest terms, the CPU takes instructions from a program/application and performs a calculation. It first *fetches* the instruction from RAM, *decodes* what the instruction actually is, and then executes the instruction using relevant parts of the CPU. The CPU performs basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions in the program.

   Originally, CPUs had a single processing core. Today's modern CPU consists of multiple cores that allow it to perform multiple instructions at once, effectively cramming several CPUs on a single chip. Almost all CPUs sold today are at least dual-core or quad-core. Additionally, a physical CPU core can perform two lines of execution (threads) at once with a process called *multithreading.* The clock speed should also be noted with CPUs: the gigahertz figure quoted on the CPU. It denotes how many instructions a CPU can handle per second (giga=billions, mega=millions).

4. The **Graphic Processing Unit (GPU)** is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images intended for output to a display device.

5. The **kernel** is a computer program at the core of a computer's operating system that has complete control over everything in the system. It facilitates interactions between hardware and software components. On most systems, the kernel is one of the first programs loaded on startup. It handles the rest of startup as well as memory, peripherals, and input/output (I/O) requests from software, translating them into data-processing instructions for the central processing unit.

   $$CPU, Memory, Devices \iff Kernel \iff Applications$$

   The critical code of the kernel is usually loaded into a separate area of memory, which is protected from access by application programs or other, less critical parts of the operating system. The kernel performs its tasks, such as running processes, managing hardware devices such as the hard disk, and handling interrupts, in this protected **kernel space**. In contrast, application programs like browsers, word processors, or audio or video players use a separate area of memory, **user space**.

All components of a computer communicate through a circuit board called the **motherboard**.

Note that all of these parts work in conjunction. For the CPU to function, it still must feed to specialized hardware the numbers they need to function. It needs to tell the graphics card to show an explosion or tell the hard drive to transfer a document to the system's RAM for quicker access.

Definition 1.2.2. Other parts of the computer include:

1. The **heatsink** is a passive heat exchanger that transfers heat. It is typically a metallic part which can be attached to a device releasing energy in the form of heat, with the aim of dissipating that heat to a surrounding fluid in order to prevent the device overheating.

Certain microcomputers with all these parts (but not packaged) include the **Raspberry Pi** and the **Arduino**.

Definition 1.2.3. A computer processes bits of information based off of how much electricity is traveling through a wire. Since there can be varying amounts of electricity running through a wire, engineers use the transistor as a "switch" that turns on when the voltage running through the wire is greater than the *threshhold voltage*.

For example, a transistor with a threshhold voltage of 4.5V will turn on when there is a current of at least 4.5V running through the wire and off otherwise.

Definition 1.2.4. A **daemon** is a type of program on Unix-like operating systems that runs unobtrusively in the background, rather than under direct control of a user, waiting to be activated.

1.3 The Internet

The **Internet** is a global network of computing devices communicating with each other in some way, whether they're sending emails, downloading files, or sharing websites. The Internet is an **open network**, which means that any computing device can join as long as they follow the **protocols** (rules that define how each device must communicate with each other). The internet is powered by many layers of protocols, and to create a global network of computing devices, we need:

1. **Wires & Wireless**: Physical connections between devices, plus protocols for converting electromagnetic signals into binary data.
2. **IP**: A protocol that uniquely identifies devices using IP addresses and provides a routing strategy to send data to a destination IP address.
3. **TCP/UDP**: Protocols that can transport packets of data from one device to another and check for errors along the way.
4. **TLS**: A secure protocol for sending encrypted data so that attackers can't view private information.
5. **HTTP & DNS**: The protocols powering the World Wide Web

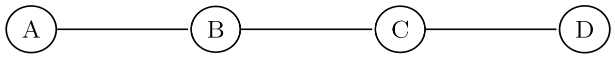
1.3.1 Computer Networks and Types of Networks

Definition 1.3.1. A **computer network** is a group of computers (i.e. computing devices) that use a set of common *communication protocols* over digital interconnections for the purpose of sharing resources located on or provided by the *network nodes*.

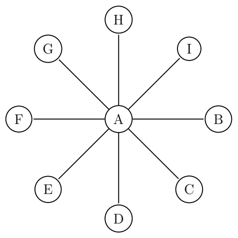
A **communication protocol** is a system of rules that allow multiple entities of a communications to transmit information via any kind variation of a physical quantity. The protocol defines the rules, syntax, semantics and synchronization of communication and possible error recovery methods.

A computer network can be visualized as a connected graph of nodes (which may include personal computers, servers, networking hardware, or other specialised or general-purpose hosts). The **network topology** is the layout, pattern, or organizational hierarchy of the interconnection of network hosts, in contrast to their physical or geographic location. Common layouts are:

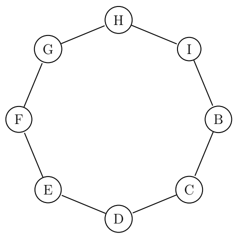
1. **Line Network**: All nodes are connected in a line.



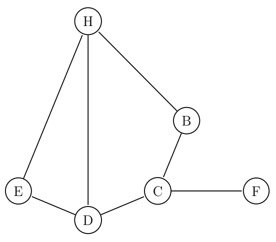
2. **Bus Network**: All nodes are connected to a common medium along this medium.
3. **Star Network**: all nodes are connected to a special central node.



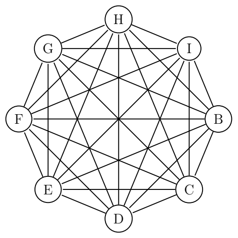
4. **Ring Network**: Each node is connected to its left and right neighbour node, such that all nodes are connected and that each node can reach each other node by traversing nodes left- or rightwards.



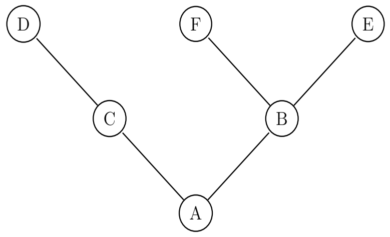
5. **Mesh Network**: each node is connected to an arbitrary number of neighbours in such a way that there is at least one traversal from any node to any other.



6. **Fully Connected Network**: each node is connected to every other node in the network.

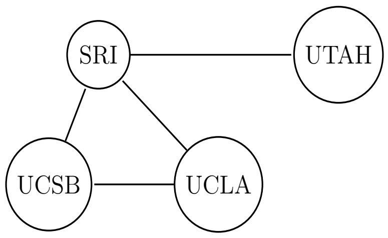


7. **Tree Network**: nodes are arranged hierarchically.



Notice how many of these networks have **redundancy**: having multiple ways to get from one node to another. That is, when a network path is no longer available, data is still able to reach its destination through another path. Usually, we would like to avoid a **single point of failure** and construct a **fault-tolerant** system that can experience failure in its components but still continue operating properly. However, building more connections may be expensive.

Example 3. The ARTPANET was the precursor to the Internet, the network where Internet technology was first tested out. It was started in 1969 with four computers connected to each other.



For example, even if the path between SRI and UCSB is gone, the connections between SRI and UCSB is not lost (since IP packets can travel through UCLA's router).

Because there are multiple paths that a piece of data takes to get from point X to point Y, *routing strategies* are implemented in order to determine the most optimal path.

Definition 1.3.2. Networks can be categorized as such:

1. A **local area network (LAN)** is a computer network that interconnects computers within a limited area. *Ethernet* and *Wi-Fi* are the two most common technologies in use for local area networks. A **wireless local area network (WLAN)** use radio frequencies to transmit and receive data.

2. A **metropolitan area network (MAN)** is a computer network that interconnects users with computer resources in a geographic region of the size of a metropolitan area.

3. In contrast, a **wide area network (WAN)** not only covers a larger geographic distance, but also generally involves *leased telecommunication circuits* or *data lines* (i.e. a private line between multiple locations provided according to a commercial contract), since no single company owns all the infrastructure across the wide geographic area. It is often composed of many LANs.

4. Another type of network is the **Data Center Network (DCN)**, a network used in data centers where data must be exchanged with very little delay.

1.3.2 Communication with Line Coding

Computers can connect through **physical** (e.g. cables) or **wireless** connections.

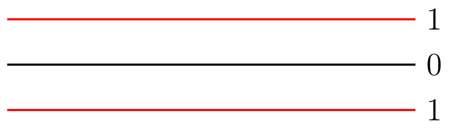
1. The **CAT5 cable** is a *twisted pair (copper) cable* that's designed for use in computer networks. It consists of four twisted pairs of copper wires. These twisted pair cables send data through a network by transmitting pulses of electricity that represent binary data. The information transmission follow the **Ethernet** standards, which is why twisted pair cables are commonly known as Ethernet cables. Use for both LANs and WANs. They can carry up to 1 Gbps across hundreds of feet, but are susceptible to interference.

2. **Fiber-optic cables** carry light instead of electricity in a fiber coated with plastic layers. The pulses of light represent binary data and also follow the Ethernet standards. They are also capable of transmitting much more data per second that copper cables, and they have the advantage of low transmission loss and immunity to electrical interference. Often used to connect networks across oceans so that data can travel quickly around the world. They can carry up to 26 Tbps acorss 50 miles (but are expensive)

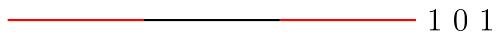
3. A wireless card inside a computer turns binary data into **radio waves** and transmits them through the air. However, they do not travel very far (100 ft in office buildings or up to 1000 ft in an open field). The waves are picked up by a *wireless access point* which converts them from radio waves back into binary data. These access points would be connected to the rest of the network using physical wiring. They can carry up to 1.3 Gbps.

4. **Infrared signals** and **microwaves** are sometimes used.

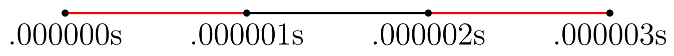
In order for the computers to send data into binary, they must convert this data into binary and send them as streams of 1s and 0s in a process called **line coding**. Furthermore, computers can raise efficiency of each wire by sending changing electric currents through a single wire. For example, rather than using three wires to encode `101` as



they send it through a single wire with intervals of $\frac{1}{3}$ seconds



or even better, at a rate of 1 megabit per second (interval of 0.000001 seconds)



As long as two computers agree on the time period in which the electricity intervals are being sent, they can communicate much more efficiently. In an electrical connection (such as Ethernet), the signal would be a voltage or current. In an optical connection (such as a fiber-optic cable), the signal would be the intensity of light.

Definition 1.3.3. There are many properties about line coding that are relevant:

1. The **bit rate** describes the data transfer rate of a connection. It measures the number of bit states that a channel can *transmit* per unit time. It is measured in *bits per second.* We can interpret it as the amount of water flowing through a pipe.

   Bit rate is typically seen in terms of the actual data rate. But for most transmissions, the data represents part of a more complex protocol, which includes bits representing source address, destination address, error detection/correction codes, and other information. This data is called the **overhead**, while the actual data transferred is called the **payload**. At times, the overhead may be substantial (up to 20% to 50%).

2. The **throughput** is the number of bit states of usable information, that can be successfully *received* over a channel per unit time. Without any channel noise, it is really just the payload. Note that this is an *observed, dynamic parameter* with a fixed and variable loss. It is also known as **consumed bandwidth** and is measured in *bits per second.*

3. The **bandwidth** describes the *maximum* data transfer rate of a connection; that is, the maximum throughput of a communication. It is measured in *bits per second.* We can interpret it as how thick the pipe is (i.e. how much water can flow through it at max). Note that this is different from the bandwidth used in signal processing.

   Data often flows over multiple network connections, which means the connection with the smallest bandwidth (most likely your local connection) acts as a bottleneck.

4. The **latency**, or **ping-rate**, measures the round trip time between the sending of a data message to a computer and the receiving of that message, measured in *milliseconds.* We can interpret it as the speed at which the water is flowing through a pipe. We can check latency by doing

   ```
   >>>ping www.google.com
   64 bytes: icmp_seq=0 ttl=115 time=37.868 ms
   ```

   which outputs a latency time of 37.868ms (to get to `www.google.com` and back) for sending a data packet of 64 bytes. Note that there is an intrinsic limiting factor to latency: the speed of light, which is approximately 1 foot per nanosecond. In addition to distance, another limiting factor is the congestion in the network and the type of connection.

Example 4. Given two computers connected by a wire that is configured to transfer 1000 bits per second, the bit rate would be 1 Kbps. However, if the channel has noise and demands retransmission of 10 bits out of every 1000 of the original transmission, then the throughput would be 990 bps.

Furthermore, the Ethernet frame can have as many as 1542 bytes. Say that there are 1500 bytes of payload and an overhead of 42 bytes. Then, the **protocol efficiency** would would be

$$\frac{\text{payload}}{\text{frame size}} = \frac{1500}{1542} = 0.9727 = 97.3\%$$

Typically, the actual line rate is stepped up by a factor influenced by the overhead to achieve an actual target net data rate. In One Gigabit Ethernet, the actual line rate is 1.25 Gbits/s to achieve a net payload throughput of 1 Gbit/s. In a 10-Gbit/s Ethernet system, gross data rate equals 10.3125 Gbits/s to achieve a true data rate of 10 Gbits/s. The net data rate also is referred to as the throughput, or payload rate, of effective data rate.

Some common units of measurement are:

1. Mbps, Gbps, Tbps (Mega, Giga, Terabits per second)
2. MBps, GBps, TBps (Mega, Giga, Terabytes per second)

In conclusion, speed is a combination of bandwidth and latency. Even if a computer is on a connection with high bandwidth, its speed of sending and receiving messages will still be limited by the latency of the connection.

1.3.3 Internet Protocol: Addresses and Routing Strategies

The **Internet Protocol (IP)** is one of the core protocols in the layers of the internet. It is used in all Internet communication to handle both addressing and routing.

Definition 1.3.4. The protocol describes the use of **IP addresses** to uniquely identify Internet-connected devices (for transmission of data). That is, when a computer sends a message to another computer, it must specify the recipient's IP address and also include its own IP address so that the second computer can reply. There are two versions of the Internet Protocol in use today:

1. **IPv4**: The first version ever used on the Internet and having the form of 4 *octets* split by periods in between.

   $$[0-255].[0-255].[0-255].[0-255]$$

   Even though it presented in decimal, computers store them in binary

   $$74.125.20.113 \iff 01001011.01111101.00010100.01110001$$

   IPv4 addresses can take 2^{32} values, but IPv6 was created for more space.

2. **IPv6**: The newer standard (introduced in June 2012) is in the form

   FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF

   with hexadecimal digits (total of 3.4×10^{39} possible IPv6 values).

Definition 1.3.5. A **dynamic IP address** is an IP address that can change. For example, each Internet service provider (ISP) has a range of addresses that they can assign, and they might give you a different one of those addresses each time your computer pops up on the network. Therefore, switching to a different WiFi network will definitely give you a new IP address.

Computers that act like servers often have **static IP addresses**. That makes it easier for computers to quickly send requests to the servers.

Definition 1.3.6 (Hierarchy of IP Addresses)**.** The IP addresses are formatted in an *hierarchical way.* The IPv4 address hierarchy is structured as such: The first few numbers (may or may not be divided by octets) could identify a **network** administered by an Internet Service Provider. The last numbers, which can also represent **subnetworks** (subnets), identifies a home computer on that network. For example, if we represent the IP address 141.213.127.13 in binary (of 32 bits)

10001101.11010101.01111111.00001101

the first 16 bits could route to all of UMich, the next two bits could route to a specific UMich department, and the final 14 bits could route to individual computers.

1000110111010101	01	11111100001101
UMich Network	Medicine department	Lab computer

This hierarchy gives UMich the ability to differentiate between 2^2 departments and $2^{14} = 16,384$ computers within each department. In general, the ability to create hierarchical levels at any point in the IP address allows for greater flexibility in the size of each level of the hierarchy.

LAN vs WAN IP Addresses

In fact, your computer is not connected to the internet directly. It is actually in a **private network**, or a *LAN network*, which uses a private IP address space (supported by both IPv4 and v6). Anything on the inside of your private network is not on the Internet; it is on your LAN, an entirely separate network, with its own address space. Anything on your LAN must have a unique (within the LAN) IP address to participate properly with your local network. Therefore, anyone else who has a LAN is also not part of the internet. Even though none of your devices in your network have a public IP address, the router itself does have a public address. That is, to the outside world, all devices identify their internet activity by the one IP address assigned by your ISP.

Definition 1.3.7. The IP addresses that are in the private network's space are usually divided up into 3 categories. But as of now, the categories don't mean anything.

1. **Class A private range addresses**: 10.0.0.0 - 10.255.255.255 (16,777,216 IPs)
2. **Class B private range addresses**: 172.16.0.0 – 172.31.255.255 (1,048,576 IPs)
3. **Class C private range addresses**: 192.168.0.0 – 192.168.255.255 (65,536 IPs)

Since the private IPv4 address space is relatively small, many private IPv4 networks unavoidably use the same address ranges. This can create a problem when merging such networks, as some addresses may be duplicated for multiple devices. In this case, networks or hosts must be renumbered, often a time-consuming task, or a network address translator must be placed between the networks to translate or masquerade one of the address ranges.

Definition 1.3.8 (NAT)**.** In order for LAN devices to connect to the Internet, their outgoing traffic has the source address changed to match that of the internet/WAN IP address of the router. The router keeps track of this, and makes sure any response traffic gets sent to the right internal machine. This is called **Network Address Translation (NAT)**. There are generally two types of NAT:

1. **Basic, one-to-one NAT**: The simplest type of NAT provides a one-to-one translation of IP addresses. In this type of NAT, only the IP addresses, IP header checksum, and any higher-level checksums that include the IP address are changed. Basic NAT can be used to interconnect two IP networks that have incompatible addressing.
2. **One-to-many NAT**: The majority of network address translators map multiple private hosts to one publicly exposed IP address. In a typical configuration, a local network uses one of the designated private IP address subnets. A router in that network has a private address of that address pace. The router it also connected to the Internet with a *public* address assigned by the ISP. As traffic passes from the local network to the Internet, the source address in each packet is translated on the fly from a private address to the public address. The router tracks basic data about each active connection (particularly the destination address and port). When a reply returns to the router, it uses the connection tracking data it stored during the outbound phase to determine the private address on the internal network to which to forward the reply.

All IP packets have a source IP address and a destination IP address. Typically packets passing from the private network to the public network will have their source address modified, while packets passing from the public network back to the private network will have their destination address modified. To avoid ambiguity in how replies are translated, further modifications to the packets are required, such as TCP or UDP.

You can find your WAN IP address simply by googling it, since your computer sends a message to the Google computers as soon as it loads `google.com`.

Example 5. In my case, my computer's LAN IP address is 192.168.0.8, my phone's LAN IP address is 192.168.0.20, and the public IP address (of the router) is 211.109.203.135. Running `whois` on my computer and phone's IP addresses reveals nothing much about the LAN one (since it is private anyways and many people have the same one), while the public one (about my router) reveals that it is owned by the ISP SK Broadband Co Ltd. But note that both the LAN and WAN IP addresses can change over time depending on your ISP (and as you restart your device).

However, if the cell phone has an active data plan and connected to a local LAN at the same time, it will have a *LAN IP address* and an *IP assigned by the mobile network.* When a browser on the phone fetches a web page, the web server will see the *LAN's gateway (public) IP address*, so essentially the phone deals with three IP's at that point. The computer on the other hand will only be assigned a LAN IP address but will also be requesting web pages via the same public IP address.

Additionally, moving my computer to another network, say Coffeebay, will change its LAN

IP address. As I am writing this in the cafe, the new IP address is now 192.168.0.168. The WAN IP address of the Coffeebay network is 125.132.4.126, with the ISP Korea Telecom (KT) Corp.

The three most popular ISPs in Korea are:

1. KT Corp.
2. LGU
3. SK Broadband

These ISPs are also called **broadband providers**. The most popular ones in the USA are:

1. AT&T Internet Services
2. Comcast High Speed Internet (aka Xfinity)
3. Verizon High Speed Internet
4. Charter Communications (including Spectrum formerly Time Warner Cable)

By typing in the correct IP into the browser, going into the administrator interface, and logging in, you can modify the WiFi settings.

Routing IP Packets

Since there are physical limitations on how large a message can be when *routing* data between computers, many networking protocols split each message into multiple small **packets**.

Definition 1.3.9. Due to physical limitations on how large a message can be sent, the Internet Protocol splits messages into **IP packets**. Each IP packet contains both a header (20 or 24 bytes long) and data (variable length).

1. The **header** includes the following:

Version/Length, Service Type, Packet Length	1 byte, 1 byte, 2 bytes
Identification	2 bytes
DF, MF, Fragment Offset	2 bytes
Time to Live, Transport	1 byte, 1 byte
Header Checksum	2 bytes
Source IP Address	4 bytes
Destination IP Address	4 bytes
Options, Padding (optional)	3 bytes, 1 byte
Total	24 bytes

2. The **data** is the actual content, such as a string of letters or part of a webpage.

These packets hop from router to router towards their destination.

Definition 1.3.10. A **router** is a type of computing device used in computer networks that helps move data packets along. The process of a router receiving and sending a packet is as such:

1. The packet gets sent to a router, and the router receives it.
2. The router looks at the packet's IP header, more specifically the destination IP address. For example, the destination IP address may be `91.198.174.192`.
3. The router must now forward the packet, but it may have multiple routers to forward it to. In order to choose the router that is the "closest" to the IP destination, it has a **forwarding table** that helps it pick the next path based on the destination IP address. The table consists of not IP addresses, but the IP address prefixes. For example,

IP address prefix	path
91.112	# 1
91.198	# 2
192.92	# 3

   Since IP addresses are by definition hierarchical, the router only needs to store the prefixes. Once the router locates the most specific row on the table for the destination IP address, it sends the packet along that path.
4. This is repeated for the next router.
5. When the final router is reached, it should have in its forwarding table the exact IP address prefix.

IP address prefix	path
91.112	# 1
91.198.174.192	Direct
192.92	#2

   This router now sends the message to the destination IP address, which may be a personal computer or a server.

Definition 1.3.11. The **modem**, short for **modulator demodulator**, "modulates" the signals going between the LAN and Internet.

The main difference between the router and the modem is that:

1. The router crates a network between the computers in your home and routes network traffic between them (through Ethernet cables or wireless connection). Your home router has one connection to the Internet and connections to your private local network.
2. The modem serves as a bridge between your local network and the Internet.

Some ISPs offer a modem and router in a single device, which has advantages and disadvantages.

Note that there are problems with these packets:

1. A computer might send multiple messages to a destination, and the destination needs to identify which packets belong to which message.
2. Packets can arrive out of order. That can happen especially if two packets follow different paths to the destination.

3. Packets can be corrupted, which means that for some reason, the received data no longer matches the originally sent data.
4. Packets can be lost due to problems in the physical layer or in routers' forwarding tables.
5. Similarly, packets might be duplicated due to accidental retransmission of the same packet.

However, there are higher level data transport protocols in the Internet protocol stack can deal with these problems, such as the *Transmission Control Protocol (TCP)* and *User Datagram Protocol (UDP)*.

1.3.4 UDP and TCP

The **User Datagram Protocol (UDP)** is a lightweight data transport protocol that works on top of IP. UDP provides a mechanism to detect corrupt data in packets, but it does not attempt to solve other problems that arise with packets, such as lost or out of order packets. That's why UDP is sometimes known as the *Unreliable Data Protocol*. UDP is simple but fast, at least in comparison to other protocols that work over IP. It's often used for time-sensitive applications (such as real-time video streaming) where speed is more important than accuracy.

When sending packets using UDP over IP, the data portion of each IP packet is formatted as a **UDP segment**.

$$\text{IP Packet} \implies \text{UDP Segment}$$

Each UDP segment contains an 8-byte header and variable length data. The first 4 bytes of the UDP header store the **port numbers** (identification numbers) for the source and destination. The next two bytes store the segment length and the checksum.

Source port #	2 bytes
Destination port #	2 bytes
Segment length #	2 bytes
Checksum #	2 bytes
Total	8 bytes

Definition 1.3.12. A networked device can receive messages on different virtual **ports**. The different ports help distinguish different types of network traffic.

For example, the following command shows the ports in use:

```
>>> sudo lsof -i -n -P | grep UDP
launchd        1            root   30u  IPv4 UDP *:137
launchd        1            root   31u  IPv4 UDP *:138
mDNSRespo    164  _mdnsresponder    6u  IPv4 UDP *:5353
mDNSRespo    164  _mdnsresponder    7u  IPv6 UDP *:5353
rapportd     356          mbahng   12u  IPv4 UDP *:3722
systemsta    665            root   17u  IPv4 UDP *:*
netbiosd   14582        _netbios    3u  IPv4 UDP *:137
netbiosd   14582        _netbios    4u  IPv4 UDP *:138
```

Each row start with the name of the process that's using the port (one for going in, another for going out) and ends with the protocol and port number. Since each port number is 2 bytes, the maximum port number can be $2^{16} = 65,535$.

The **segment length** stores the length of the the entire UDP segment (including the header). Since the segment length can be represented in 16 bits, the maximum length of the UDP segment can be 65,635 bytes.

The **checksum** is used check for data corruption. It is computed as such. Before sending off the segment, the sender:

1. Computes the checksum based on the data in the segment (by literally summing up sections of the the binary representation of the data.
2. It stores the computed checksum in the field.

Upon receiving the segment, the recipient:

1. Computes the checksum based on the received segment.
2. Compares the checksums to each other. If the checksums aren't equal, it knows that the data was corrupted.

The **Transmission Control Protocol (TCP)** is a transport protocol that is used on top of IP to ensure reliable transmission of packets. TCP includes mechanisms to solve many of the problems that arise from packet-based messaging, such as lost packets, out of order packets, duplicate packets, and corrupted packets. Since TCP is the protocol used most commonly on top of IP, the Internet protocol stack is sometimes referred to as **TCP/IP**.

When sending packets using TCP/IP, the data portion of each IP packet is formatted as a **TCP segment**.

$$\text{IP Packet (4 bytes)} \implies \text{TCP Segment (variable)}$$

The TCP header can contain many more fields than the UDP header and can range in size from 20 to 60 bytes, depending no the size of the options field. It does contain the source port number, destination port number, and checksum.

Source port #	2 bytes
Destination port #	2 bytes
Sequence number	4 bytes
Acknowledgement #	4 bytes
Offset and Reserved	10 bits
URG, AFK, PSH, RST, SYN, FIN bits	6 bits
Window Size	2 bytes
Checksum	2 bytes
Urgent pointer	2 bytes
Options/Padding	4 bytes
Total	39 bytes

The process of transmitting a packet with TCP/IP is as such:

1. Two computers first establish a connection through a **three-way handshake**. Computer 1 sends a packet with the SYN bit set to 1. Then computer 2 sends back a packet with the ACK bit set to 1 plus the SYN bit set to 1. The first computer replies back with ACK=1. (Note that the SYN and ACK bits are part of the TCP header) The three packets involved in the three-way handshake do not typically include any data. Once the computers are done with the handshake, they're ready to receive packets containing actual data.

2. When a packet of data is sent over TCP, the recipient must always acknowledge what they received in the following way: Computer 1 sends a packet with data and a sequence number. Computer 2 acknowledges it by setting the ACK bit and increasing the acknowledgement number by the length of the received data (note that both numbers are also part of the TCP header). It is easy to see how these two numbers help the computers keep track of which data was successfully received, which data was lost, and which data was accidentally sent twice.

3. To close the connection, computer 1 initiates it by sending a packet with the FIN bit set to 1. Computer 2 replies with an ACK and another FIN. After one more ACK from computer 1, the connection is closed.

However, some problems can occur. TCP connections can detect lost packets using a timeout. After sending off a packet, the sender starts a timer and puts the packet in a retransmission queue. If the timer runs out and the sender has not yet received an ACK from the recipient, it sends the packet again. The retransmission may lead to the recipient receiving duplicate packets, if a packet was not actually lost but just very slow to arrive or be acknowledged (which happens when the packet takes a slower route through the Internet). If so, the recipient can simply discard duplicate packets.

TCP connections can also detect out of order packets by using the sequence and acknowledgement numbers. When the recipient sees a higher sequence number than what they have acknowledged so far, they will know that they are missing at least one packet in between.

1.3.5 Domain Name System (DNS), Hypertext Transfer Protocol (HTTP)

Definition 1.3.13. The **world wide web**, or **the web**, is a network of webpages, programs, and files that are accessible via URLs. It is a subsection of the Internet. A web browser loads a webpage using various protocols:

1. **Domain Name System (DNS) protocol** for converting domain names into IP addresses.

2. **HyperText Transfer Protocol (HTTP)** to request the webpage contents from that IP address.

3. **Transport Layer Security (TLS) protocol** to serve the website over a secure, encrypted connection.

Note that the web browser uses these protocols on top of the Internet protocols. The Web is just one of the applications built on top of the Internet protocols, but it is by far the most popular.

DNS

Definition 1.3.14. Computers are identified by their IP addresses, but to make these addresses more readable by humans, we identify them through the **domain name system**. For example, `www.wikipedia.org` connects us to the computers powering Wikipedia. Each domain name is made up of parts:

`third-level-domain.second-level-domain.top-level-domain`

1. There are a limited set of top level domains (**TLD**s), and many websites use the most common TLDs, such as `.com`, `.org`, and `.edu`.
2. The second level domain is unique to the company or organizaiton that registers it, like `wikipedia` or `facebook`.
3. The third level domain, also called a subdomain, is also owned by the same group of the second level domain. It often just directs you to a subset of the website.

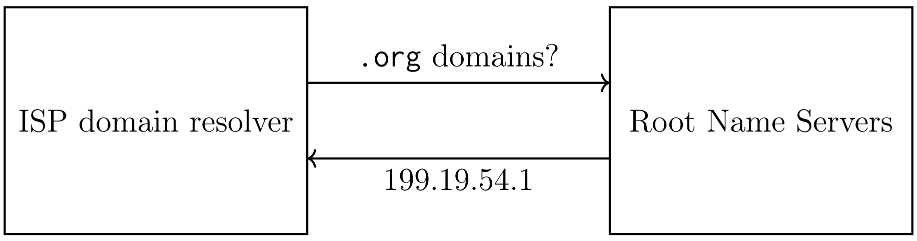
$$\begin{aligned} \texttt{m.wikipedia.org} &\implies \text{mobile-optimized Wikipedia} \\ \texttt{es.khanacademy.org} &\implies \text{Spanish-language Khan Academy} \end{aligned}$$

In reality, these domain names are only for humans, and each domain name maps to an IP address. But since the computer can't store the 300 million domain names locally, it goes through a multi-step process to find out the IP address.

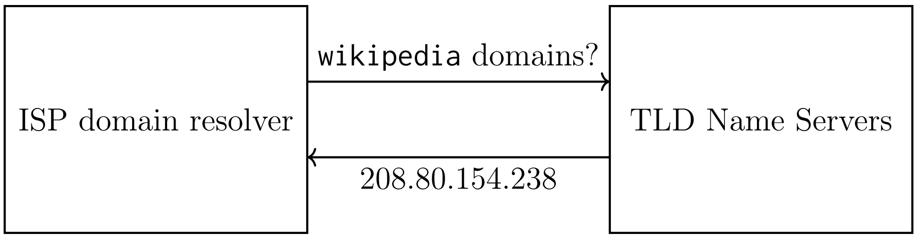
1. Check the local cache. Since people often visit the same website multiple times, they keep their own local cache of domain name to IP mappings. The cache stays small and is constantly updated. Each browser can keep their own cache.
2. Ask the ISP (Internet Service Provider) cache. Every ISP provides a domain name resolving service and keeps its own cache so the cache may contain the websites accessed by people with the same ISP (such as neighbors).
3. Ask the name servers. There are domain name servers scattered around the globe that are responsible for keeping track of a subset of the millions of domain names. There are three types of servers, which are ordered in a hierarchy:

   $$\text{Root name servers} \rightarrow \text{TLD name servers} \rightarrow \text{Host name servers}$$

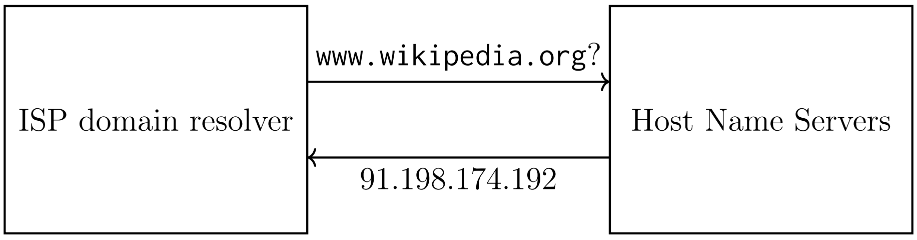
   The ISP starts by going to the root name servers and sending a request for the IP address of, say the name server of the `.org` domains. The root name server responds with the IP address of a TLD name server that tracks `.org` domains.

   Then, the ISP asks the TLD name server for, say the `wikipedia` domains. The TLD name server responds with the IP address of the host name server that contains the wikipedia records.



Finally, the ISP asks the host name server for the specific domain `www.wikipedia.org`. The host name server responds with an exact IP address, and now the computer can successfully connect with the computer powering that domain.



However, lots of information is cached, so it is rare that a **DNS lookup** has to go through all the steps.

However, the domain name system is not always secure. For example, if a cyber-criminal manages to take control of a name server or redirect requests to its own server, then it can have the server give out false IP addresses that can lead to websites filled with malware. This act is called **DNS spoofing**, or **DNS cache poisoning**. Fortunately, the recent **DNSSEC protocol** extends the original DNS protocol and specifies the best way for DNS resolvers to authenticate the information sent to them, which can prevent DNS spoofing.

HTTP

Whenever a pageon the web is visited, the computer uses the **Hypertext Transfer Protocol** to download that page from another computer somewhere on the Internet. The steps of this process are as such:

1. We access the web with a **browser application**. The user either types a **Uniform Resource Locator (URL)** in the browser or follows a link from an already opened page.

2. The domain names of these URLs map to IP addresses, the true location of the domain's computers. The browser uses a DNS resolver to map the domain to an IP address.

3. Browser sends HTTP request. Once the browser identifies the IP address of the computer hosting the requested URL (i.e. the **host computer**), it sends an **HTTP request**. An example HTTP request can be:

```
GET /index.html HTTP/1.1
Host: www.example.com
```

   (a) The word `GET` is the request. There are other verbs for other actions, such as `POST` for submitting form data.

(b) The next part specifies the path (`/index.html`). The host computer stores the content of the entire website, so the browser needs to be specific about which page to load.

(c) The final part of the first line specifies the protocol and the version of the protocol: `HTTP/1.1`.

(d) The second line specifies the domain of the requested URL. That's helpful in case a host computer stores the content for multiple websites.

4. Once the host computer receives the HTTP request, it sends back a **HTTP response** with both the content and metadata about it.

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 208
<!DOCTYPE html>
<html>
    <head>
        <title>Example Domain</title>
    </head>
    <body>
        <h1>Example Domain</h1>
        <p>This domain is to be used for illustrative examples in
                                          documents.</p>
    </body>
</html>
```

(a) The `HTTP/1.1` is the protocol and version. The next number is the **HTTP status code**. In this case, a `200` represents a successful retrieval of the document, "OK." Another code is the `404` code, which represents "file not found."

(b) The 2nd and 3rd lines are the **headers**, which provides additional details. The content-type tells the browser what type of document is being sent back. `text/html` represent HTML text files; `image/png` represent images; `video/mpeg` are videos; `application/javascript` are scripts; and so on. The content length gives the length of the document in bytes.

(c) The rest of the HTTP response writes our the actual document requested.

5. The browser renders the response, and you see the regular webpage.

Note that HTTP is a protocol that is built on top of the TCP/IP protocols. That is, each HTTP request/response is inside an IP packet (or more often, in multiple packets). There are many other protocols built on top of TCP/IP, like protocols for sending email (SMTP, POP, IMAP) and uploading files (FTP).

Note that the protocols powering the Internet and the Web were designed for scalability. Any computing device can send data around the Internet if it follows the protocols, and routing is dynamic, so new routers can join a network at any time and help to move data packets around the internet.

Definition 1.3.15. A **scalable system** is one that can continue functioning well even as it experiences higher usage.

However, there are limitations to the scalability of the internet. For example,

1. Network connections have limited bandwidth, so huge amounts of data could easily overwhelm low bandwidth connections, leading to delays or dropped packets.
2. Routers have limited throughput (the amount of data they can forward per second). A modern consumer router has a throughput around 1 Gbps while much more expensive enterprise routers can forward up to 10 Gbps.
3. Wireless routers often have a limitation in the number of devices that can be connected to them, typically up to 250 devices. If everyone tried to use a shared WiFi network at the same time (like in a university or library), they might find themselves simply unable to join.

Definition 1.3.16. Engineering teams can prepare for spikes in usage by doing **load testing**: simulating high amounts of traffic in a short period of time to stress test the system. Load testing can uncover bottlenecks or hard-coded limits in the system.

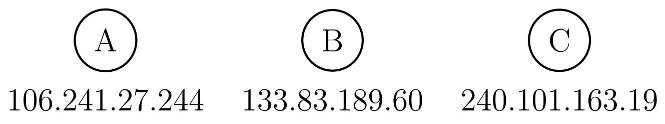
1.3.6 The Internet Protocol Suite

There are many protocols that power the Internet. Each protocol operates at a different layer, building functionality on top of the layer below it. We can visualize it in the following diagram.

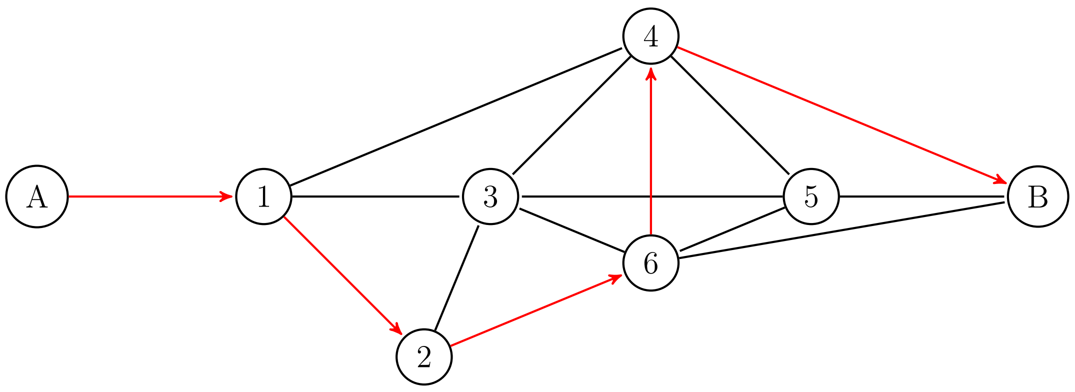
Application Layer	HTTP, TLS, DNS
Transport Layer	TCP, UDP
Network Layer	IP (v4, v6)
Link Layer	Ethernet, Wireless LAN

1. At the link layer, 2 computing devices need a physical mechanism to send digital data to each other. They send electromagnetic signals either over a wired or wireless connection and interpret the signal as bits. This type of physical connection affects the bit rate and bandwidth.

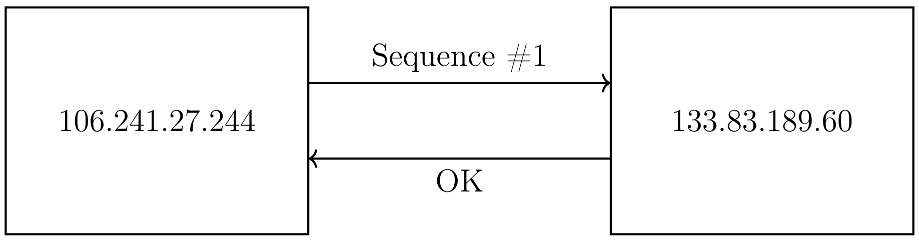
2. Once a network is bigger than two computers, we need addressing protocols to uniquely identify who is sending data and who should receive the data. Every node on the Internet is identified with an IP address.



3. The route between any two computers on the Internet isn't just a straight path from A to B. The data must pass from router to router until it finally reaches its destination, a strategy that comes from the Internet routing protocol.



4. Data needs to be broken up into small packets, which are then reassembled at the destination. The Transmission Control Protocol (TCP) is used to ensure reliable transport of those packets, with sequencing, acknowledgement, and retries. A faster but less reliable transport protocol is the User Datagram Protocol (UDP).



The Transport Layer Security (TLS) protocol uses algorithms to encrypt the data (cryptography). *Certificate authorities* help users trust the encryption.

Example 6. When loading a webpage from a domain your browser has never visited before, your browser may need to make a DNS request, which is represented by the following stack of protocols when the request is sent through the internet.

Application Layer	DNS
Transport Layer	UDP
Network Layer	IP
Link Layer	Wireless LAN

Then, your browser will make an HTTP request to fetch the webpage. This prototol stack is used when an HTTP request is sent:

Application Layer	HTTP
Transport Layer	TCP
Network Layer	IP
Link Layer	Wireless LAN

If the webpage is served over HTTPS, then the stack includes multiple protocols at the application layer (both HTTP and TLS):

Application Layer	HTTP, TLS
Transport Layer	TCP
Network Layer	IP
Link Layer	Wireless LAN

In a network, we must also make sure that the computers follow the *same* protocol, i.e. there is *standardization* within the network. Fortunately, the protocols of the Internet are **open** (not owned by any particular company and not limited to a particular company's products). For every protocol that is both standardized and open, there is a publicly viewable document describing the protocol, often called the **specification**. These specifications are maintained by the **Internet Engineering Task Force (IEFT)**. Some of them are:

1. the HTTP specification
2. the TCP specification

The languages of the web are also open standards with online specifications. They include the

1. HTML living standard
2. many specifications for CSS
3. ECMAScript standard for JavaScript.

Checking Network Processes

To check all open network connections (by process) and their usage of network bandwidth, use the `nettop` keyword on the command line. This can be used to find out which processes are taking up all of your local network bandwidth.

```
>>>nettop
                            interface       state     bytes_in   bytes_out
systemstats.64                                             0 B         0 B
udp4 *:*<->*:*

config.66                                                  0 B         0 B
udp4 *:*<->*:*

remoted.71                                             489 KiB     680 KiB
tcp6 IPv6\%en3.49160<->*.*        en3       Listen
tcp6 IPv6\%en3.49160<->IPv6       en3  Established     494 KiB     687 KiB

apsd.97                                                 18 KiB      21 KiB
tcp4 IPv4:52737<->IPv4:5223       en0  Established      18 KiB      21 KiB

timed.99                                                 144 B       144 B

usbmuxd.100                                            104 KiB      75 KiB
tcp4 IPv4:52695<->IPv4:54848      en0  Established     104 KiB      75 KiB

bluetooth.119                                              0 B         0 B
udp4 *:*<->*:*

AirPlayXPCHelpe.123                                        0 B         0 B
udp4 *:*<->*:*
udp4 *:*<->*:*

loginwindow.131                                            0 B         0 B
udp4 *:*<->*:*
```

```
mDNSResponder.164                                      6996 KiB   2606 KiB
udp6 *.5353<->*.*                  lo0                 1717 KiB    790 KiB
udp4 *.5353<->*:*                  en0                 5235 KiB   1808 KiB

symptomsd.189                                             0 B          0 B
udp4 *:*<->*:*

findmydeviced.190                                       1349 B      3061 B
tcp6 IPv6\%en3.49161<->IPv6.49248 en3 Established       1349 B      3061 B

airportd.198                                              0 B          0 B
udp4 *:*<->*:*
udp4 *:*<->*:*
udp4 *:*<->*:*
udp6 *:*<->*:*
udp4 *:*<->*:*

corekdld.203                                             293 B      31 KiB
tcp6 IPv6\%en3.49163<->IPv6.49256 en3 Established        293 B      31 KiB

bosUpdateProxy.204                                       406 B      2903 B
tcp6 IPv6\%en3.51336<->IPv6.49260 en3 Established        406 B      2903 B

#Only the processes will be shown from now on.

SubmitDiagInfo.205                                     65 KiB        9210 B
mobileactivatio.206                                    9238 B       11 KiB
locationd.270                                             0 B          0 B
biometrickitd.290                                     509 KiB       95 KiB
accountsd.338                                          5518 B        1247 B
trustd.342                                             4363 B         610 B
Simplenote.355                                         6316 B        6661 B
rapportd.356                                           10 KiB        4965 B
ControlCenter.359                                         0 B          0 B
Finer.361                                                 0 B          0 B
identityservice.369                                    4600 B        2961 B
itunescloudd.374                                       30 KiB        2533 B
com.apple.geod.392                                     3512 B        1353 B
WirelessRadioMa.414                                       0 B          0 B
sharingd.417                                              0 B          0 B
nsurlsessiond.420                                       6 KiB        9246 B
CalendarAgent.424                                      95 KiB       39 KiB
wifivelocityd.426                                         0 B          0 B
NewsToday2.437                                        392 KiB       27 KiB
AMPDeviceDiscov.441                                    2602 B        3725 B
assistantd.453                                         22 KiB        8764 B
ScreenTimeWidge.463                                    8029 B         936 B
WeatherWidget.464                                      7471 B        1168 B
corespeechd.471                                        7226 B       25 MiB
com.apple.Safar.530                                    11 KiB        2478 B
AdobeDesktop S.616                                        0 B          0 B
node.640                                                  0 B          0 B
comapple.WebKi.1274                                    17 MiB      627 KiB
com.appleSafar.1335                                    9956 B        2303 B
PowerChime.1353                                        2853 B       19 KiB
adprivacyd.1388                                        24 KiB        5253 B
lskdd.1768                                             1805 B        5851 B
PerfPowerServic.15327                                     0 B          0 B
```

```
netbiosd.19203                                    277 KiB        142 KiB
bluetoothaudiod.19268                               0 B            0 B
Notify.20116                                        0 B            0 B
```

Note that pressing `q` quits nettop; pressing `p` renders the traffic numbers as bytes or in human-readable formats; pressing `c` collapses the display, showing only the network apps (no sockets); pressing `e` expands the display to show sockets.

Under each process (network application) there is a list of **sockets**, which are endpoints in a two-way communication between two programs on a network; for example, between a web server and your web browser. Looking at the columns,

1. The leftmost column contains the list of all the names of the processes and sockets, followed by a dot and their process ID (PID); so in the form

   `NetworkApp.PID`

   For example, `locationd.270` would be the process locationd with a PID of 270.

2. A **network interface** is the point of interconnection between a computer and a private/public network. The *network interface card* connects your computer to a local data network or the internet. The card translates computer data into compatible electrical signals it sends through the network.

   (a) `lo0` is loopback interface. A **loopback** is the routing of electronic signals, digital data streams, or flows of items back to their source without intentional processing or modification. It is primarily a means of testing the transmission tests.

   (b) `en0` is Wifi (was ethernet at one point)

   (c) `fw0` is the FireWire network interface

   (d) `utun1`

   (e) `stf0` is an IPv6 to IPv4 tunnel interface to support the transition from IPv4 to IPv6 standard.

   (f) `gif0` is a more generic tunneling interface.

   (g) `awdl0` is Apple Wireless Direct Link

3. The **state** refers to the state of the connection between sockets.

   (a) The state of a server waiting for a connection on a port is `Listen`

   (b) The state of a connection recently closed is `TimeWait`

   (c) `Established` means that the connection is active

   (d) `SynSent` occurs when a client initiates the connection to a server by sending the SYN packet (a part of the 3-way handshake in TCP) and awaits the ACK packet.

4. The `bytes_in` and `bytes_out` shows how much traffic has come in and gone out for that socket (or for the entire process).

The format of each socket is

```
TransportProtocolVersion localhostIPaddress:port<->remoteIPaddress:port
```

Notice that all of the sockets use the standardized UDP or TCP protocol (with the corresponding IP address Version: IPv4 or IPv6).

1. `udp4, udp6` - the connection is one-way since there are no SYN and ACK bits being sent to confirm the connection.
2. `tcp4, tcp6` - the connection guarantees that both ends are aware of one other, so datagrams can be sent back and forth until the FIN bit is sent and acknowledged.

A transport protocol (say, tcp) that use IPv4 is in the form

```
tcp4 192.168.0.88:52737<->17.57.145.138:5223
```

while one that uses IPv6 is in the form

```
tcp6 fe80::aede:48ff:fe00:1122%en3.4915<->fe80::aede:48ff:fe33:445%en3.5960
```

Some sockets may have asterisks rather than actual IP addresses in them.

```
airportd.198                                                      0 B        0 B
udp4 *:*<->*:*
udp4 *:*<->*:*
udp6 *:*<->*:*
```

An asterisk means that these sockets are open. The operating system creates these open sockets as placeholders of sorts, so that it can respond faster to incoming data (since incoming data will trigger the creation of a socket, which causes delay).

Definition 1.3.17. In addition to the IP address, the **port number** is the part of the addressing information used to determine what protocol incoming traffic should be directed to. That is, port number identifies a specific process to which an Internet or other network message is to be forwarded when it arrives at a server. They are represented by 16-bit numbers, meaning that port numbers can have values up to $2^{16} = 65,536$. However,

1. 0-1023 are restricted port numbers and are used by well-known protocol services. Some of them include:
   (a) 80 for HTTP
   (b) 123 for NTP
   (c) 67, 68 for DHCP traffic
   (d) 443 for HTTPS (almost all ports in the browser socket will be 443).
   (e) 137, 138 for netbios. NetBIOS is a protocol used for File and Print Sharing under all current versions of Windows.
2. 1024-49,151 are registered port numbers; they can be registered to specific protocols by software corporations.
3. 49,152-65,536 are used as dynamic/private ports and can be used by anybody.

The differences between the IP address and port number are:

1. The IP address is used to identify a host in the network, while a port number is used to identify a process/service on your system.
2. IP address is the address of the layer-3 Internet protocol suite, while the port number is the address of the layer-4 protocols.
3. IP address is provided by admin of system or network administrator, while the port number is provided by the kernel of the operating system.

1.3.7 Online Data Security

Definition 1.3.18. Personally identifiable information (PII) refers to data that can directly or indirectly identify individuals. Some of the most common PIIs are:

1. Name
2. Social Security Number
3. Biometric Data (DNA, fingerprints, etc.)

Another weaker form of PII are **linkable PII**, which refers to data that can be combined from separate sources to identify individuals.

It is hard to classify certain data as PII or not since the capabilities and the creative use of them is changing. An instance of when attackers steal PII from companies is known as a **data breach**. You can check whether you are a victim of a data breach with services like `haveibeenpwned.com`.

The web is not private by default; websites often use **cookies** to track user action on their site and even across other sites (to improve their services).

Definition 1.3.19. An **HTTP cookie** is a small amount of text that helps a website track information about a user across multiple pages of the website and personalize the user's experience on the website.

If you've ever logged into a website, a cookies kept you logged in across multiple pages. A cookie is set in the following steps:

1. When a user navigates to a website for the first time (in a particular browser), the browser sends an HTTP request to the server that hosts the website.

```
GET /index.html HTTP/1.1
Host: www.shoopshop.com
```

2. The server sends back an HTTP response and includes a `Set-cookie` header in that response.

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: sessionId=abc123; Expires=Wed, 09 Jun 2021 10:18:14
                                GMT
...
```

The cookie contains a name (sessionId) and a value (abc123), plus an expiration date for the browser to clear this cookie from its memory. If it wants to set multiple cookies, it adds more Set-cookie headers to the response.

3. The browser saves the cookie information, storing it on the user's hard drive. That way, the data will persist even after restarting the browser or computer, which is why this type of cookie is called a **persistent cookie**. There are also **session cookies** which have no expiration date and are always deleted when the browser is shut down.

4. When the user navigates to a different page on the website, the browser sends along the stored cookies with each HTTP request.

```
GET /shop.html HTTP/1.1
Cookie: sessionId=abc123
```

5. When the server receives the HTTP request, it inspects the cookies and sees that this request is coming from a user with a known sessionId. It can then look up that session ID in its database and use any information about the session to personalize the response.

Cookies can have many uses, such as:

1. A search engine can use them to remember how many results a user prefers seeing per page.

2. A news site can use them to recommend headlines that are similar to the articles you've already read.

3. All sorts of websites can use cookies to track analytics, like how long you spent on a page and which buttons you clicked.

4. Any website with a log-in uses a cookie to keep you logged in on every page of the site. When you log out of that site, it clears the cookie and doesn't set it again until you login again.

It is clear that you should never share your cookies.

Definition 1.3.20. Each cookie stored by a browser is associated with a domain and path. When you visit a website and its server sends back an HTTP response with a cookie, the browser associates that cookie with the domain of the server. That's called a **first-party cookie**.

However, a website can also include resources from other domains, like an image, iframe, or script. When the browser requests those resources, their servers can also send back cookies, which will now be associated with their domain. These are called **third-party cookies**.

Imagine a user that visits a food blog with a recipe for gluten-free cookies. That blog includes a Facebook ad with a cookie. The user then visits facebook.com and notices a sudden uptick in ads about gluten-free products, which resulted from the cookies in the Facebook ad in the blog. Third-party cookies more often serve the purpose of collecting information for advertising and infringe more on the privacy of web users.

Search and Browsing History

Definition 1.3.21. A **search engine** is a service that builds an index of the World Wide Web and gives users a way to search that index.

It is important to know that in order to improve their services (like spelling correction) all search engines collect data on search queries (i.e. what was searched). A search query itself isn't private information, but some search engines can log much more than the query:

Search query	Date	Time	IP address	User agent
"jet ski"	03/11/20	11:14	49.121.111.73	Mozilla/5.0 (Windows NT 5.1)
"home depot"	03/11/20	16:00	49.121.111.73	Mozilla/5.0 (Windows NT 5.1)
"cheap pizza"	03/12/20	21:07	49.121.111.73	Mozilla/5.0 (Windows NT 5.1)
"Windsor"	03/13/20	14:32	49.121.111.73	Mozilla/5.0 (Windows NT 5.1)

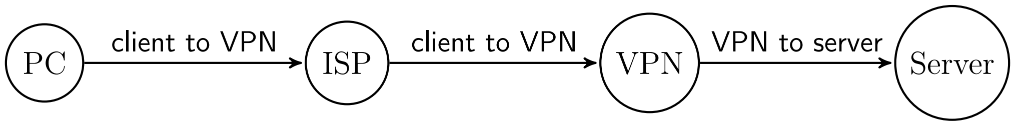
A combination of these queries over a period of time can definitely be PII, and additionally, the search history can include a cookie or even a user ID if you were logged into the search engine website when you issued the query. Third party cookies can allow a website to track a user's browsing history across other websites, as long as each site loads the cookie from the same domain.

Many browsers also provide an *incognito browsing mode*, which will not store browsing history at all. Once you close the window, it will also forget any cookies generated in that session. There are certain search engines, such as DuckDuckGo, that collect only search queries and do not collect any PII.

Note that since all requests (packets of data) are forward through the router, anyone with access to the router can monitor the destinations of HTTP requests. An Internet Service Provider (ISP) administers the first routers that a packet travels through (excluding the home/office/school) router, so the ISP can see every HTTP request that's sent through those routers. Users can use HTTPS-secured websites to hide the contents of their requests, but HTTPS will still reveal the domain names. ISPs can use that information to find customers that are engaged in illegal activities, such as downloading pirated movies. Government organization such as the National Security Agency (NSA) have reportedly installed backdoor surveillance monitoring programs on routers before they were exported to foreign customers.

VPNs and Tor

Definition 1.3.22. When using a **Virtual Private Network (VPN)**, the computer sends a packet of encrypted data with a destination of the VPN server to the ISP. The VPN server decrypts the data, finds out where the user actually wants to send the packet, and then forwards the packet to that destination.



The VPN server knows the user's browsing history, but the ISP does not. Plus, other routers after the VPN will only see that the packet came from the VPN IP address, not from the user's IP address. A VPN subscription is often expensive, however, and the additional stop along the way can result in a slower browsing experience.

Another option is **Tor**, an open source program for anonymizing Internet traffic. When using Tor, the computer sends an encrypted packet through a large number of volunteer relays. The data is packaged such that each relay only knows where it came from and where it's going, and no relay knows both the sender IP address and the destination IP address.

Tor can provide truly anonymous browsing, but it also severely slows down the browsing experience, since it has to hop through volunteer relays that can be located anywhere on the Internet.

Definition 1.3.23. A **proxy server** is a server application or appliance that acts as an intermediary for requests from clients seeking resources from servers that provide those resources. A proxy server thus functions on behalf of the client when requesting service, potentially masking the true origin of the request to the resource server.

Instead of connecting directly to a server that can fulfill a requested resource, such as a file or web page, the client directs the request to the proxy server, which evaluates the request and performs the required network transactions. This serves as a method to simplify or control the complexity of the request, or provide additional benefits such as load balancing, privacy, or security. Proxies were devised to add structure and encapsulation to distributed systems. Some types of proxies are:

1. A **gateway** or a **tunneling proxy** is a proxy server that passes unmodified requests and responses.
2. An **open proxy** is a forwarding proxy server that is accessible by any Internet user. Hundreds of thousands of open proxies are operated on the internet.
   (a) **Anonymous proxies** reveals its identity as a proxy server, but does not disclose the originating IP address of the client.
   (b) **Transparent proxies** also identifies itself as a proxy server, but the originating IP address can be retrieved. The main benefit of using this type of server is its ability to cache a website for faster retrieval.
3. A **reverse proxy** is a proxy server that appears to clients to be an ordinary server. Reverse proxies forward requests to one or more ordinary servers that handle the request. The response from the proxy server is returned as if it came directly from the original server, leaving the client with no knowledge of the original server. Reverse proxies aer installed in the neighborhood of one or more web servers, and all traffic coming from the Internet goes through the proxy server. The use of *reverse* originates in its counterpart *forward proxy* since the reverse proxy sits closer to the web server and serves only a restricted set of websites.
   (a) Encryption/SSL acceleration: When secure websites are created, the *Secure Sockets Layer (SSL)* encryption is often not done by the web server itself, but by a reverse proxy that is equipped with SSL acceleration hardware.
   (b) Load balancing: the reverse proxy can distribute the load to several web servers, each web server serving its own application area.

(c) Serve/cache static content: A reverse proxy can offload the web servers by caching static content like pictures and other static graphical content.

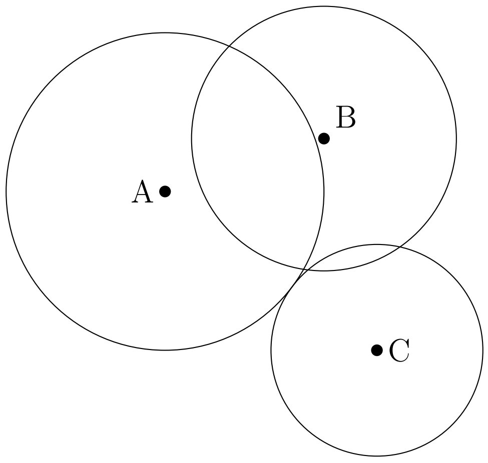
(d) Compression: the proxy server can optimize and compress the content to speed up the load time.

(e) Security: the proxy server is an additional layer of defense and can protect against some OS and Web Server specific attacks. However, it does not provide any protection from attacks against the web application or service itself, which is generally considered the larger threat.

Geolocation

Definition 1.3.24. The **geolocation** of a device is an approximate latitude and longitude describing its geographic location.

Definition 1.3.25. The most popular method in which geolocation is determined is through **trilateration**, which is a geometric process of of determining absolute or relative locations of points by measurements of distances, using the geometry of circles, spheres ,or triangles. For example, given three points $A, B, C \in \mathbb{R}^2$, a unique point with certain distances from A, B, C can be determined.



Other methods of trilateration exist.

One way to determine geolocations is through the **Global Position System (GPS)**, a project started by the US government in the 1970s controlled by approximately 30 GPS satellites orbiting the Earth. **GPS receivers** are tiny sensors with antennas that receive radio signals from the GPS satellites orbiting in the sky above. If a sensor can receive signals from at least 4 satellites, the receiver can calculate its position using trilateraion. Since they depend on radio signals from satellites, GPS is most accurate in an outdoor environment with a clear view of the sky.

On the other hand, **WiFi positioning** is a strategy that works well in dense, urban areas filled with WiFi networks. First, a device with a WiFi antenna scans for WiFi access points and measures the signal strength to each network. (Note that signal strength is always negative, so the number closest to 0 is strongest)

BSSID	MAC address	Signal strength (RSSI)
NETGEAR09	A3:F3:5D:2A:A3:1B	-59
NETGEAR09-5G	A3:F3:5D:2A:A3:1B	-72
Sonic-b346	53:19:DA:E0:57:3A	-79
Emdutos	E3:84:14:BC:BC:FF	-84
Baskind Bunch	52:8D:5E:29:E7:5A	-85
Sonic-9472-5G	4C:4C:DB:91:1A:1A	-88
xfinitywifi	F8:59:F4:FC:C5:F1	-93

Then, the device determines the location of each access point by looking it up in a WiFi location database or in their own (smaller) cache of locations. It then estimates its own location based on the found locations and their signal strength using trilateration.

A more accurate technique is **fingerprinting**, but only possible if a fingerprint map has been made ahead of time. To make the map, a portable device computes the fingerprint for many reference points within a particular area. Each fingerprint is the list of nearby networks and their signal strength, like the table above, plus a pair of geographic coordinates. When a mobile device enters the area and needs to know its location, it can send its fingerprint to the machine with the radio map, and the machine uses an algorithm to compute the closest fingerprint and estimate the coordinates accordingly. This is basically just using WiFi positioning ahead of time.

If a cell phone is unable to use GPS to report its location, it can instead use **cell tower trilateration**. Cell towers are what makes cellular networks possible. Each cell tower includes three sets of directional antenna arrays in a triangular shape, and using trilateration, multiple cell towers can be used to determine the geolocation of a mobile device.

The least accurate of all methods is **IP-based geolocation**. IP geolocation databases contain millions of rows mapping IP addresses to locations. Companies create those databases based on a variety of sources such as regional IP address registries, user-submitted locations on websites, data from ISPs, and estimates based on network routes. They usually get the country and state correct, but often there are deviations in any more specific location data. Furthermore, if a user is accessinfg the Internet through a VPN, their true IP will be hidden and the VPN's IP could be geolocated in an entirely different continent.

Note that when a user visits a website, their browser sends an HTTP request to the web server. The HTTP request is wrapped in an IP packet, so it always includes the sender's IP address. Therefore, the web server can always use an IP geolocation service to turn the user's IP address into an approximate location, which can give better demographics for the company.

Cyber Attacks

A **phishing attack** is an attempt to trick a user into divulging their private information. Some signs of a phishing attack are:

1. suspicious email addresses. However, a legitimate email address is not a guarantee that an email is 100% safe. Attackers mught have figured out a way to spoof the legitimate email address or hacked their way into control over the actual email.

2. Suspicous URL. Attackers may
   (a) misspell the original URL (`goggle.com`)
   (b) use similar looking characters from other alphabets (the `e` and `a`) in `wikipedia.org` are actually different characters in those two domains)
   (c) have subdomains that look like the domain name. (`paypal.accounts.com` vs `accounts.paypal.com`)
   (d) have a different top level domain (TLD) (`paypal.io` vs `paypal.com`). Popular companies try to buy their domain with the most common TLDs, such as `.net, .com, .org`, but there are hundreds of TLDs out there.
   (e) Have a hyperlinked text directed to a different URL.
3. Phishing websites sometimes may not use HTTPS. Any website that is asking you for sensitive information should be using HTTPS to encrypt the data sent over the Internet.

Definition 1.3.26. An **access point** acts as a translator between wireless and wired signals. Access points connect to the Internet via a wired connection but share it wirelessly with many devices like your computer. Most routers include access points since they are responsible for transporting packets, not for providing wireless Internet access. Most of them have an Ethernet cable in the back that connects it to the Internet and antennae that broadcase and receive wireless signals.

However, another form of cyberattacking is through **rogue access points**, which an access point installed on a network without the network owner's permission. If an attacker owns the access point, they can intercept the data (PII) flowing through the network. There are two ways rogue access points can intercept PII:

1. *Passive interception.* A rogue access point can read your data but cannot manipulate it. If you connect to a network with a rogue access point and enter your password on a site over HTTP, the rogue access point can read your password. They also have access to your Internet footprint.
2. *Active interception.* In active interception, a rogue access point can also manipulate your data. They can read the incoming user data, modify the data however they want, and send the modified user data to the destination endpoint. For example, if a user visits a banking website and tries to deposit money into an account, a rogue access point can redirect the deposit to an attacker's account.

We can also protect ourselves by using VPNs (virtual private networks) or HTTPS. VPNs and HTTPS both send an encrypted form of our data across the network. Even if rogue access points intercept it, they won't be able to unscramble it.

Definition 1.3.27. **Malware** is malicious software that is unknowningly installed onto a computer and often tries to steal personal data or make money off the user.

1. A **trojan horse** is a harmful program that masquerades as a legitimate program.
2. A **virus** is self-replicating: it contains code that copies itself into other files on the system. Viruses may hide in the code of a legitimate program.

3. A **worm** is also self-replicating, but copies itself into entirely different computers within the network. It can travel along networked protocols such as email, file sharing, or instant messaging.

The effects of malware are:

1. **Spyware** steals data and sends it back to the malware creators. A common form of spyware are keyloggers, programs that monitor everything a user types including passwords.

2. **Adware** pops up advertisements to users.

3. **Ransomware** holds a computer hostage y encrypting user data or blocking access to applications, and it demands the user pay a ransom to the anonymous malware creators.

4. **Cryptomining malware** utilizes a computer's resources to mine for cryptocurrency. That allows the creators to earn cryptocurrency without needing to spend money on powering their own computers.

Some protection mechanisms against malware include:

1. A security patch is an update to the code of an application or the entire operating system, and often fixes a bug that's been exploited by malware.

2. A **firewall** is a system that monitors incoming and outgoing network traffic to a computer or internal network, and determines what traffic to allow. Firewalls can do automated detection of suspicious traffic and can also be configured manually.

3. Antivirus software protects an individual computer by constantly scanning files and identifying malware. Once an antivirus program finds a piece of malware, it can guide the user through deleting or repairing the file to be safe again.

Secure Internet Protocols

We assume that the reader is familiar with basic encryption techniques, including public-key encryption.

Definition 1.3.28. A **symmetric encryption** is any technique where the same key is used to both encrypt and decrypt the data (e.g. the Caesar Cipher, Vigenere Cipher).

The **Transport Layer Security (TLS)** adds a layer of security on top of the TCP/IP transport protocols, using both symmetric and public key encryptionfor securely sending private data. Even though this extra process increases latency in Internet communications, the security benefits are well worth it. The process is described as such:

1. TCP handshake. The client must first complete the 3-way TCP handshake with the server.

2. TLS initiation. Then, the client must notify the server that it desires a TLS connection instead of the standard insecure connection, so it sends a message describing which TLS protocol version and encryption techniques it'd like to use.

3. Server confirmation of protocol. If the server doesn't support the client's requested technologies, it will abort the connection. That may happen if a modern client

is trying to communicate with an older server. As long as the server does support the requested TLS protocol version and other options, it will respond with a confirmation, plus a digital certificate that contains its public key.

4. Certificate verification. The client can verify (or choose not to) the certificate. The client now knows the public key of the server, so it can theoretically use public key encryption to encrypt data that the server can then decrypt with its corresponding private key. For speed, they use a combination of public-key and symmetric encryption to share data.
5. With this, the client can securely send private data to the server, using symmetric encryption and the shared key.

Notice that all of this depends on the credibility of the **digital certificate** which proves the ownership of an encryption key. A server that wants to communicate securely over TLS signs up with a **certificate authority (CA)**. The certificate authority verifies their ownership of the domain, signs the certificate with their own name and public key, and provides the signed certificate back to the server. This question now boils down to whether the certificate authority can be trusted, and there are actually intermediate CAs that verifies other CAs. The CA at the "top" of this chain that verifies is called the *root CA*.

With standard HTTP, many people can see what we're reading on the Internet, which is why websites are increasingly using **HTTPS (Hypertext Transfer Protocol Secure)** to protect the privacy of their uses and prevent tampering. HTTPS is also known as HTTP over TLS, because it's implemented by encrypting HTTP requests and responses with the TLS protocol.

When the browser loads a URL that starts with `https`, it begins the process of setting up a secure connection over TLS. Early in that process, the browser must verify the digital certificate of the domain. If the browser cannot verify the certificate, then the browser may display a certificate error (e.e. the message "Your connection is not private"). If the certificate is valid, most browsers will display a lock in the address bar, which indicates a secured connection over HTTPS.

An HTTPS connection ensures that only the browser and the secured domain see the data in HTTP requests and responses. Onlookers can still see that a particular IP address is communicating with another domain/IP and they can see how long that connection lasts. But those onlookers can't see the content of the communication, which includes the full URL path, the webpage HTML, and any text submitted in forms.

1.4 Representation of Data

The inputs of a computational program at its most fundamental level really takes in a **binary string** of 0s and 1s. Note that the choice of 0 and 1 is for convenience, but it must be binary (i.e. Boolean) in some way in order for the model to be physically implemented by transistors. Once information is in digital form, we can *compute* over it and gain insights from data that were not accessible in prior times. In fact, we can represent an unbounded variety of objects using only two symbols 0 and 1.

Therefore, when we say that a program P takes x as an input, we really mean that P

takes as input the *representation of* x as a binary string.

Definition 1.4.1. A *representation scheme* is a way to map an object x to a unique binary string $E(x) \in \{0,1\}^*$. That is, given a set of objects, E is an injective (not not necessarily surjective) map

$$E : X \longrightarrow \{0,1\}^*$$

1.4.1 Representation of Numbers

Definition 1.4.2 (Representation of the Naturals)**.** A representation for natural numbers (note that in this context, $0 \in \mathbb{N}$) is the (non-surjective) regular binary representation denoted

$$NtS : \mathbb{N} \longrightarrow \{0,1\}^* \quad (NtS = \text{"Naturals to Strings"})$$

recursively defined as

$$NtS(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ NtS(\lceil n/2 \rceil \, parity(n) & n > 1 \end{cases}$$

where given strings $x, y \in \{0,1\}^*$, xy denotes the concatenation of x and y, and $parity : \mathbb{N} \longrightarrow \{0,1\}^*$ is defined

$$parity(n) = \begin{cases} 0 & n \text{ is even} \\ 1 & n \text{ is odd} \end{cases}$$

Since NtS in injective, its inverse $StN : \operatorname{Im} NtS \subset \{0,1\}^* \longrightarrow \mathbb{N}$ is well-defined.

Definition 1.4.3 (Representation of the Integers)**.** To construct a representation scheme for $\mathbb{Z}$, we can just add one more binary digit to represent the sign of the number. The binary representation $ZtS : \mathbb{Z} \longrightarrow \{0,1\}^*$ is defined

$$ZtS(m) = \begin{cases} 0 \, NtS(m) & m \geq 0 \\ 1 \, NtS(-m) & m < 0 \end{cases}$$

where NtS is defined as before. Again this function must be injective but need not be surjective.

When representing rational numbers, we cannot simply concatenate the numerator and denominator as such

$$a/b \mapsto ZtS(a) \, ZtS(b)$$

since this map is not surjective (and may overlap with other integers).

Definition 1.4.4 (Representation of Rationals)**.** To represent a rational number a/b, we create a separator symbol $|$ and map the rational number as below in the alphabet $\{0, 1, |\}$.

$$q : a/b \mapsto ZtS(a) | ZtS(b)$$

Then, we use a second map that goes through each digit in z and is defined

$$p : \{0, 1, |\} \longrightarrow \{00, 11, 01\} \subset \{0,1\}^2, \; p(n) = \begin{cases} 00 & n = 0 \\ 11 & n = 1 \\ 01 & n = | \end{cases}$$

Therefore, p maps the length n string $z \in \{0,1\}^*$ to the length $2n$ string $\omega \in \{0,1\}^*$. The representation scheme for $\mathbb{Q}$ is simply

$$QtS \equiv p \circ q$$

Example 7. Given the rational number $-5/8$,

$$\frac{-5}{8} \mapsto 1101|01000 \mapsto 11110011010011000000$$

This same idea of using separators and compositions of injective functions can be used to represent arbitrary n-tuples of strings (since a finite Cartesian product of countable sets is also countable).

Theorem 1.4.1 (Representation of Vectors)*.* All vectors over the field $\mathbb{Q}$ are representable.

Proof. We can simply create another separator symbol $\cdot$ and have the initial mapping q map to a string over the alphabet $\{0, 1, |, \cdot\}$, which injectively maps to $\{00, 01, 10, 11\}$. ■

Corollary 1.4.1.1 (Representation of Matrices and Tensors)*.* Matrices (over $\mathbb{Q}$), which are a collection of vectors are representable in binary. Furthermore, general tenors over the field $\mathbb{Q}$ are representable in binary.

Proof. Create more separator symbols and map them to a sufficiently large set (which can be extended arbitrarily). For example, to perhaps $\{000, 001, ..., 111\}$. ■

Corollary 1.4.1.2 (Representation of Graphs)*.* Directed graphs, which can be represented with their adjacency matrices, can therefore be represented with binary strings.

Theorem 1.4.2 (Representation of Images)*.* Every finite-resolution image can be represented as a binary number.

Proof. Since we can interpret each image as a matrix where each element (a pixel) is a color, and since each color can be represented as a 3-tuple of rational numbers corresponding to the intensities of red, green, and blue (for humans, we can restrict it to three primary colors), all images can eventually be decomposed into binary strings. ■

Theorem 1.4.3 (Representation of Reals)*.* There exists no representation of the reals

$$NtR : \mathbb{R} \longrightarrow \{0,1\}^*$$

Proof. By Cantor's theorem, the reals are uncountable. That is, there does not exist a surjective function $NtR : \mathbb{N} \longrightarrow \mathbb{R}$. The implies the nonexistence of an injective inverse; that is, there does not exist an injective function

$$RtS : \mathbb{R} \longrightarrow \{0,1\}^*$$

■

However, since $\mathbb{Q}$ is dense in $\mathbb{R}$, we can approximate every real number x by a rational number a/b to arbitrary accuracy. There are multiple ways to construct these approximations (decimal approximation up to kth digit, finite continued fractions, truncated infinite series, etc.), but computers use the *floating-point approximation.*

Definition 1.4.5 (Floating-Point Representation)**.** The **floating-point representation scheme** of a real number $x \in \mathbb{R}$ is its approximation as a number of the form

$$\sigma b \cdot 2^e$$

where $\sigma \in \{0, 1\}$ determines the sign of the representation of x, e is a (potentially negative) integer, and b is a rational number between 1 and 2 expressed as a binary fraction

$$1.b_0 b_1 b_2 ... b_k = 1 + \frac{b_1}{2} + \frac{b_2}{4} + ... + \frac{b_k}{2^k}, \quad b_i \in \{0, 1\}$$

where the number k is fixed (determined by the desired accuracy; greater k implies more digits and better accuracy). The $\sigma b \cdot 2^e$ closest to x is the *floating-point representation*, or *approximation*, of x. We can think of σ determining the sign, e the order of magnitude (in base 2) of x, and b the value of the number scaled down to a value in $[1, 2)$, called the *mantissa.*

1.4.2 Representation of General Sets

Let there exist some set $\mathcal{O}$ consisting of objects. Then, a representation scheme for representing objects in $\mathcal{O}$ consists of an *encoding* function that maps an object in $\mathcal{O}$ to a string, and a *decoding* function that decodes a string back to an object in $\mathcal{O}$.

Definition 1.4.6. Let $\mathcal{O}$ be any set. A *representation scheme for* $\mathcal{O}$ is a pair of functions E, D where

$$E : \mathcal{O} \longrightarrow \{0, 1\}^*$$

is an injective function, and the induced mapping D is restriction of the inverse of E to the image of E.

$$D : \text{Im}(E) \subset \{0, 1\}^* \longrightarrow \mathcal{O}$$

This means that $(D \circ E)(o) = o$ for all $o \in \mathcal{O}$. E is known as the *encoding function* and D is known as the *decoding function.*

Prefix-free Encoding

Definition 1.4.7 (Prefix)**.** For two strings y, y', y is a prefix of y' if y' "starts" with y. That is, y is a **prefix** of y' if $|y| \leq |y'|$ and for every $i < |y|$, $y'_i = y_i$.

With this, we can define the concept of prefix free encoding.

Definition 1.4.8. Let $\mathcal{O}$ be a nonempty set and $E : \mathcal{O} \longrightarrow \{0, 1\}^*$ be a function. E is **prefix-free** if $E(o)$ is nonempty for every $o \in \mathcal{O}$ and there does not exist a distinct pair of objects $o, o' \in \mathcal{O}$ such that $E(o)$ is a prefix of $E(o')$.

Being prefix-free is a nice property that we would like an encoding to have. Informally, this means that no string x representing an object o is an initial substring of string y representing a different object o. This means that we can simply represent a *list* of objects simply by concatenating the representations of all the list members and still get a valid, injective representation. We formalize this below.

Theorem 1.4.4. Suppose that $E : \mathcal{O} \longrightarrow \{0,1\}^*$ is prefix free. Then the following map

$$\overline{E} : \mathcal{O}^* \longrightarrow \{0,1\}^*$$

over all finite length tuples of elements in $\mathcal{O}$ is injective, where for every $o_0, o_1, ..., o_{k-1} \in \mathcal{O}^*$, we define $\overline{E}$ to be the simple concatenation of the separate encodings of o_i:

$$\overline{E}(o_0, ..., o_{k-1}) \equiv E(o_0)E(o_1)...E(o_{k-1})$$

Even if the representation E of objects in $\mathcal{O}$ is prefix free, this does not imply that our representation $\overline{E}$ of *lists* of such objects will be prefix free as well. In fact, it won't be, since for example, given three objects o, o', o'', the representation of the list (o, o') will be a prefix of the representation of the list (o, o', o'').

However, it turns out that in fact we can transform *every* representation into prefix free form, and so will be able to use that transformation if needed to represents lists of lists, lists of lists of lists, and so on.

Some natural representations are prefix free. For example, every *fixed output length* representation (i.e. an injective function $E : \mathcal{O} \longrightarrow \{0,1\}^n$) is automatically prefix-free, since a string x can only be a prefix of an equal length x' if x and x' are identical. Moreover, the approach that was used for representing rational numbers can be used to show the following lemma.

Lemma 1.4.5. Let $E : \mathcal{O} \longrightarrow \{0,1\}^*$ be a one-to-one function. Then there is a one-to-one prefix-free encoding $\overline{E}$ such that

$$|\overline{E}(o)| \leq 2|E(o)| + 2$$

for every $o \in \mathcal{O}$.

Proof. The general idea is the use the map $0 \mapsto 00$, $1 \mapsto 11$ to "double" every bit in the string x and then mark the end of the string by concatenating to it the pair 01. If we encode a string x in this way, it ensures that the encoding of x is never a prefix of the encoding of a distinct string x'. (Note that this is not the only or even the best way to transform an arbitrary representation into prefix-free form.) ■

1.4.3 Representing Letters and Text

We can represent a letter or symbol by a string, and then if this representation is prefix free, we can represent a sequence of symbols by merely concatenating the representation of each symbol. Here are a few examples.

ASCII

The **ASCII** (also called US-ASCII) code, which stands for American Standard Code for Information Interchange is a 7 bit character code where every single bit represents a unique character. ASCII codes represent text in computers, telecommunications equipment, and other devices. Most modern character-encoding schemes are based on ASCII, although they support many additional characters.

The first 32 characters are called the *control characters*: codes originally intended not to represent printable information, but rather to control devices (such as printers) that make use of ASCII, or to provide meta-information about data streams. For example, character 10 (decimal) represents the "line feed" function (which causes a printer to advance its paper) and character 8 represents "backspace." Except for the control characters that prescribe elementary line-oriented formatting, ASCII does not define any mechanism for describing the structure or appearance of text within a document.

Dec	Oct	Hex	Bin	Symbol	Description
0	000	00	0000000	NULL	Null char
1	001	01	0000001	SOH	Start of Heading
2	002	02	0000010	STX	Start of Text
3	003	03	0000011	ETX	End of Text
4	004	04	0000100	EOT	End of Transmission
5	005	05	0000101	ENQ	Enquiry
6	006	06	0000110	ACK	Acknowledgement
7	007	07	0000111	BEL	Bell
8	010	08	0001000	BS	Back Space
9	011	09	0001001	HT	Horizontal Tab
10	012	0A	0001010	LF	Line Feed
11	013	0B	0001011	VT	Vertical Tab
12	014	0C	0001100	FF	Form Feed
13	015	0D	0001101	CR	Carriage Return
14	016	0E	0001110	SO	Shift Out/X-On
15	017	0F	0001111	SI	Shift In/X-Off
16	020	10	0010000	DLE	Data Line Escape
17	021	11	0010001	DC1	Device Control 1
18	022	12	0010010	DC2	Device Control 2
19	023	13	0010011	DC3	Device Control 3
20	024	14	0010100	DC4	Device Control 4
21	025	15	0010101	NAK	Negative Acknowledgement
22	026	16	0010110	SYN	Synchronous Idle
23	027	17	0010111	ETB	End of Transmit Block
24	030	18	0011000	CAN	Cancel
25	031	19	0011001	EM	End of Medium
26	032	1A	0011010	SUB	Substitute
27	033	1B	0011011	ESC	Escape
28	034	1C	0011100	FS	File Separator
29	035	1D	0011101	GS	Group Separator
30	036	1E	0011110	RS	Record Separator
31	037	1F	0011111	US	Unit Separator

The rest of the characters are the ASCII printable characters.

Dec	Oct	Hex	Bin	Sym	Description	Dec	Oct	Hex	Bin	Sym	Description
32	040	20	0100000		Space	80	120	50	1010000	P	Uppercase P
33	041	21	0100001	!	Exclamation	81	121	51	1010001	Q	Uppercase Q
34	042	22	0100010	"	Double quotes	82	122	52	1010010	R	Uppercase R
35	043	23	0100011	#	Number	83	123	53	1010011	S	Uppercase S
36	044	24	0100100	$	Dollar	84	124	54	1010100	T	Uppercase T
37	045	25	0100101	%	Per cent sign	85	125	55	1010101	U	Uppercase U
38	046	26	0100110	&	Ampersand	86	126	56	1010110	V	Uppercase V
39	047	27	0100111	’	Single quote	87	127	57	1010111	W	Uppercase W
40	050	28	0101000	(	Open paren.	88	130	58	1011000	X	Uppercase X
41	051	29	0101001	)	Closed paren.	89	131	59	1011001	Y	Uppercase Y
42	052	2A	0101010	*	Asterisk	90	132	5A	1011010	Z	Uppercase Z
43	053	2B	0101011	+	Plus	91	133	5B	1011011	[	Opening bracket
44	054	2C	0101100	,	Comma	92	134	5C	1011100	\	Backslash
45	055	2D	0101101	-	Hyphen	93	135	5D	1011101	]	Closing bracket
46	056	2E	0101110	.	Period	94	136	5E	1011110	ˆ	Caret
47	057	2F	0101111	/	Slash	95	137	5F	1011111	_	Underscore
48	060	30	0110000	0	Zero	96	140	60	1100000	‘	Grave accent
49	061	31	0110001	1	One	97	141	61	1100001	a	Lowercase a
50	062	32	0110010	2	Two	98	142	62	1100010	b	Lowercase b
51	063	33	0110011	3	Three	99	143	63	1100011	c	Lowercase c
52	064	34	0110100	4	Four	100	144	64	1100100	d	Lowercase d
53	065	35	0110101	5	Five	101	145	65	1100101	e	Lowercase e
54	066	36	0110110	6	Six	102	146	66	1100110	f	Lowercase f
55	067	37	0110111	7	Seven	103	147	67	1100111	g	Lowercase g
56	070	38	0111000	8	Eight	104	150	68	1101000	h	Lowercase h
57	071	39	0111001	9	Nine	105	151	69	1101001	i	Lowercase i
58	072	3A	0111010	:	Colon	106	152	6A	1101010	j	Lowercase j
59	073	3B	0111011	;	Semicolon	107	153	6B	1101011	k	Lowercase k
60	074	3C	0111100	<	Less than	108	154	6C	1101100	l	Lowercase l
61	075	3D	0111101	=	Equals	109	155	6D	1101101	m	Lowercase m
62	076	3E	0111110	>	Greater than	110	156	6E	1101110	n	Lowercase n
63	077	3F	0111111	?	Question mark	111	157	6F	1101111	o	Lowercase o
64	100	40	1000000	@	At symbol	112	160	70	1110000	p	Lowercase p
65	101	41	1000001	A	Uppercase A	113	161	71	1110001	q	Lowercase q
66	102	42	1000010	B	Uppercase B	114	162	72	1110010	r	Lowercase r
67	103	43	1000011	C	Uppercase C	115	163	73	1110011	s	Lowercase s
68	104	44	1000100	D	Uppercase D	116	164	74	1110100	t	Lowercase t
69	105	45	1000101	E	Uppercase E	117	165	75	1110101	u	Lowercase u
70	106	46	1000110	F	Uppercase F	118	166	76	1110110	v	Lowercase v
71	107	47	1000111	G	Uppercase G	119	167	77	1110111	w	Lowercase w
72	110	48	1001000	H	Uppercase H	120	170	78	1111000	x	Lowercase x
73	111	49	1001001	I	Uppercase I	121	171	79	1111001	y	Lowercase y
74	112	4A	1001010	J	Uppercase J	122	172	7A	1111010	z	Lowercase z
75	113	4B	1001011	J	Uppercase K	123	173	7B	1111011	{	Opening brace
76	114	4C	1001100	L	Uppercase L	124	174	7C	1111100	|	Vertical bar
77	115	4D	1001101	M	Uppercase M	125	175	7D	1111101	}	Closing brace
78	116	4E	1001110	N	Uppercase N	126	176	7E	1111110	∼	Tilde
79	117	4F	1001111	O	Uppercase O	127	177	7F	1111111		Delete

Extended ASCII

The **Extended ASCII** (EASCII or high ASCII) character encodings are 8-bit or larger encodings that include the standard 7-bit ASCII characters, plus additional characters. Note that this does not mean that the standard ASCII coding has been updated to include more than 128 characters nor does it mean that there is an universal extension to the original ASCII coding. In fact, there are several (over 100) extended ASCII encodings.

With the creation of the 7-bit ASCII format, increased need for more letters and symbols (such as characters in other languages or more punctuation/mathematical symbols). With better computers and software, it became obvious that they could handle text that uses 256-character sets at almost no additional cost in programming or storage. The 8-bit format would allow ASCII to be used unchanged and provide 128 more characters.

But even 256 characters is still not enough to cover all purposes, all languages, or even all European languages, so the emergence of *many* ASCII-derived 8-bit character sets was inevitable. Translating between these sets (*transcoding*) is complex, especially if a character is not in both sets and was often not done, producing **mojibake** (semi-readable text resulting from text being decoded using an unintended character encoding. The result is a systematic replacement of symbols with completely unrelated ones, often from a different writing system). ASCII can also be used to create graphics, commonly called **ASCII art**.

But ASCII isn't enough. We have lots of languages with lots of characters that computers should ideally display. Unicode assigns each character a unique number, or code print. Computers deal with such numbers as bytes: 8-bit computers would treat an 8-bit byte as the largest numerical unit easily represented on the hardware, 16-bit computers would expand that to 2 bytes, and so forth. Old character encodings like ASCII are from the (pre-) 8-bit era, and try to cram the dominant language in computing at the time, i.e. English, into numbers ranging from 0 to 127 (7 bits). When ASCII got extended by an 8th bit for other non-English languages, the additional 128 numbers/code points made available by this expansion would be mapped to different characters depending on the language being displayed. The **ISO-8859** standards are the most common forms of this mapping:

1. **ISO-8859-1**
2. **ISO-8859-15**, also called **ISO-Latin-1**

But that's not enough when you want to represent characters from more than one language, so cramming all available characters into a single byte just won't work. The following shows ways to do this (that is compatible with ASCII).

ISO-10646, UCS

We can simply expand the value range by adding more bits. The UCS-2 uses 2 bytes (or 16 bits) and UCS-4 uses 4 bytes (32 bits). However, these codings suffer from inherently the same problem as ASCII and ISO-8859 standards, as their value range is still limited, even if the limit is vastly higher. Note that these encode from the ISO-10646, which defines several character encoding forms for the Universal Coded Character Set.

1. UCS-2 can store $2^{16} = 65,536$ characters.
2. UCS-4 can store $2^{32} = 4,294,967,296$ characters.

Notice that UCS encoding has a fixed number of bytes per character, which means that UCS-2 stores each character in 2 bytes, and UCS-4 stores each character in 4 bytes. This is different from **UTF-8** encoding.

ISO 10646 and Unicode have an identical repertoire and numbers—the same characters with the same numbers exist on both standards, although Unicode releases new versions and adds new characters more often. Unicode has rules and specifications outside the scope of ISO 10646. ISO 10646 is a simple character map, an extension of previous standards like ISO 8859. In contrast, Unicode adds rules for collation, normalization of forms, and the bidirectional algorithm for right-to-left scripts such as Arabic and Hebrew. For interoperability between platforms, especially if bidirectional scripts are used, it is not enough to support ISO 10646; Unicode must be implemented.

Unicode, UTF-8

Unicode is the universal character encoding, maintained by Unicode Consortium, and it covers the characters for all the writing systems of the world, modern and ancient. It also includes technical symbols, punctuation, and many other characters used in writing text. As of Unicode Version 13.0, the Unicode standard contains 143,859 characters, stored in the format U+****, where **** is a number in hexadecimal notation. Notice that these ones are not fixed in the number of bits; that is,

U+27BD and U+1F886

are perfectly viable representations of characters in Unicode. Even though only 143,859 characters are in use, Unicode currently allows for 1,114,112 ($16^5 + 16^4$) code values, and assigns codes covering nearly all modern text writing systems, as well as many historical ones and for many non-linguistic characters such as printer's dingbats, mathematical symbols, etc.

*Note that Unicode, along with ISO-10646, is a standard that assigns a name and a value (**Character Code** or **Code-Point**) to each character in its repertoire.* However, the Unicode format must be encoded in a binary format for the computer to understand. When you save a document, the text editor has to explicitly set its encoding to be UTF-8 (or whatever other format) the user wants it to be. Also, when a text editor program reads a file, it needs to select a text encoding scheme to decode it correctly. Even further, when you are typing and entering a letter, the text editor needs to know what scheme you use so that it will save it correctly. Therefore, *UTF-8 encoding is a way to represent these characters digitally in computer memory.* The way that **UTF-8** encodes characters is with the following format:

```
1st Byte    2nd Byte    3rd Byte    4th Byte    Number of Free Bits
0xxxxxxx                                                  7
110xxxxx    10xxxxxx                                  (5+6)=11
1110xxxx    10xxxxxx    10xxxxxx                    (4+6+6)=16
11110xxx    10xxxxxx    10xxxxxx    10xxxxxx    (3+6+6+6)=21
```

From this, we can see that UTF-8 uses a variable number of bytes per character. All UTF encodings work in roughly the same manner: you choose a unit size, which for UTF-8 is 8 bits, for UTF-16 is 16 bits, and for UTF-32 is 32 bits. The standard then defines a few of these bits as *flags* (e.g. the 0, 110, 1110, 11110, ...). If they're set, then the next unit in a sequence of units is considered part of the same character. If they're not set, this unit represents one character fully. Thus, the most common (English) characters only occupy one byte in UTF-8 (two in UTF-16, 4 in UTF-32), but other language characters can occupy more bytes. We can see that UTF-8 can encode up to (and slightly more than) $2^{21} = 2,097,152$ characters. UTF-8 is by far the most common encoding for the World Wide Web, accounting for 96.0% of all web pages, and up to 100% for some languages, as of 2021.

For example, let's take a random character, say with the Unicode value to be U+6C49. Then, we convert this to binary to get

01101100 01001001

But we can't just store this because this isn't a prefix-free notation. This is when UTF-8 is needed. Using the chart above, we need to prefix our character with some headers/flags. The binary Unicode value of the character is 16 bits long, so we can store it in 3 bytes (in the format of the third row) as it provides enough space. The headers are not bolded, while the binary values added are.

1110**0110** 10**110001** 10**001001**

We can take another example of a character with the Unicode value U+1F886. Converting to binary gets

0001 1111 1000 1000 0110

There are 20 bits, so we will need to store it in 4 bytes (in the format of fourth row) as it provides enough space (21). We convert the 20-bit-long binary Unicode value to a 21-bit-long value (so that it is compatible with the 21 free bits) to get

0 0001 1111 1000 1000 0110

Encoding it in UTF-8 in 4 bytes gives

11110**000** 10**011111** 10**100010** 10**000110**

There is no need to go beyond 4 bytes since every Unicode value will have at most 5 hexadecimal digits (since $16^5 = 1,048,576$, which is far more than the number of characters there are). There is also another, obsolete, encoding used called the **UTF-7**.

Both the UCS and UTF standards encode the code points as defined in Unicode. In theory, those encodings could be used to encode any number (within the range the encoding supports) - but of course these encodings were made to encode Unicode code points. Windows handles so-called "Unicode" strings as UTF-16 strings, while most UNIXes default to UTF-8 these days. Communications protocols such as HTTP tend to work best with UTF-8, as the unit size in UTF-8 is the same as in ASCII, and most such protocols were designed in the ASCII era. On the other hand, UTF-16 gives the best average space/processing performance when representing all living languages.

While UTF-7, 8, 16, and 32 all have the nice property of being able to store *any* code point correctly, there are hundreds of encodings that can only store a set amount of characters. If there's no equivalent for the Unicode code point you're trying to represent in the encoding you're trying to represent it in, you usually get a little question mark: ? For example, trying to store Russian or Hebrew letters in these encodings results in a bunch of question marks.

All Plaintext depends on Encodings

Note that **it does not make sense to have a string without knowing what encoding it uses**. We can't just assume that every plaintext is in ASCII, since there are hundreds of extended ASCII encodings. If you have a string, in memory, in a file, or in an email message, you have to know what encoding it is in or you cannot interpret it or display it to users correctly.

For example, when you are sending an email, Gmail is the only client that automatically converts your text to UTF-8, regardless of what you set in the header. The browser also uses a certain encoding, which can be accessed (and changed) under the "view" tab.

1.4.4 Text Files

The ASCII character set is the most common compatible subset of character sets for English-language text files, and is generally assumed to be the default file format in many situations.

In the Mac, checking the character encoding of a text file can be done with the command

```
>>> file -I filename.txt
filename.txt: text/plain; charset=us-ascii
```

ASCII covers American English, but for the British Pound sign, the Euro sign, or characters used outside English, a richer character set must be used. In many systems, this is chosen based on the default setting on the computer it is read on. Prior to UTF-8, this was traditionally single-byte encodings (such as ISO-8859-1 through ISO-8859-16) for European languages and wide character encodings for Asian languages. However, most computers use UTF-8 as the natural extension. We can check this firsthand by inputting a non-ASCII character in filename.txt, which would result in

```
>>> file -I filename.txt
filename.txt: text/plain; charset=utf-8
```

Because encodings necessarily have only a limited repertoire of characters, often very small, many are only usable to represent text in a limited subset of human languages. Unicode is an attempt to create a common standard for representing all known languages, and most known character sets are subsets of the very large Unicode character set. Although there are multiple character encodings available for Unicode, the most common is UTF-8, which has the advantage of being backwards-compatible with ASCII; that is, every ASCII text file is also a UTF-8 text file with identical meaning. UTF-8 also has the advantage that it is easily auto-detectable. Thus, a common operating mode of UTF-8 capable software, when opening files of unknown encoding, is to try UTF-8 first and fall back to a locale dependent legacy encoding when it definitely isn't UTF-8.

Because of their simplicity, text files are commonly used for storage of information. When data corruption occurs in a text file, it is often easier to recover and continue processing the remaining contents. A disadvantage of text files is that they usually have a low entropy, meaning that the information occupies more storage than is strictly necessary. A simple text file may need no additional metadata (other than knowledge of its character set) to assist the reader in interpretation. A text file may contain no data at all, which is a case of zero-byte file.

Plain Text, Rich Text

Plain text is a loose term for data (e.g. file contents) that represent only characters of readable material but not its graphical representation nor other objects (floating-point numbers, images, etc.). It may also include a limited number of "whitespace" characters that affect simple arrangement of text, such as spaces or line breaks. A plain text file cannot have bold text, fonts, larger font sizes, or any other special text formatting.

Plain text is different from **formatted text** or **rich text**, where style information is included and structured text, where structral parts of the document such as paragraphs, sections, and the like are identified.

Most systems associate plain text file with the file .txt. To view a plaintext file, a text editor must be used.

1.4.5 Binary Files

Another type of data is a **binary file**, which is a computer file that is not a text file (it is often used as a term meaning *non-text file*). Many binary file formats contain parts that can be interpreted as text; for example, some computer document files containing formatted text, such as older Microsoft Word document files, contain the text of the document but also contain formatting information in binary form.

Binary files are usually thought of as being a sequence of bytes, which means the binary digits (bits) are grouped in eights. Binary files typically contain bytes that are intended to be interpreted as something other than text characters. Compiled computer programs are typical examples; indeed, compiled applications are sometimes referred to, particularly by programmers, as binaries. But binary files can also mean that they contain images, sounds, compressed versions of other files, etc. – in short, any type of file content whatsoever.

Viewing Binary Files

A hex editor or viewer may be used to view file data as a sequence of hexadecimal (or decimal, binary or ASCII character) values for corresponding bytes of a binary file.

If a binary file is opened in a text editor, each group of eight bits will typically be translated as a single character, and the user will see a (probably unintelligible) display of textual characters. If the file is opened in some other application, that application will have its own use for each byte: maybe the application will treat each byte as a number and output a stream of numbers between 0 and 255—or maybe interpret the numbers in the bytes as colors and display the corresponding picture. Other type of viewers (called 'word extractors') simply replace the unprintable characters with spaces revealing only the human-readable text. This type of view is useful for a quick inspection of a binary file in order to find passwords in games, find hidden text in non-text files and recover corrupted documents. It can even be used to inspect suspicious files (software) for unwanted effects. For example, the user would see any URL/email to which the suspected software may attempt to connect in order to upload unapproved data (to steal). If the file is itself treated as an executable and run, then the operating system will attempt to interpret the file as a series of instructions in its machine language

Standards are very important to binary files. For example, a binary file interpreted by the ASCII character set will result in text being displayed. A custom application can interpret the file differently: a byte may be a sound, or a pixel, or even an entire word. Binary itself is meaningless, until such time as an executed algorithm defines what should be done with each bit, byte, word or block. Thus, just examining the binary and attempting to match it against known formats can lead to the wrong conclusion as to what it actually represents. This fact can be used in *steganography*, where an algorithm interprets a binary data file differently to reveal hidden content. Without the algorithm, it is impossible to tell that hidden content exists.

Chapter 2

Information Theory

2.1 Entropy

Informally, entropy measures the uncertainty of a random variable.

Definition 2.1.1. Let X be a discrete random variable with alphabet $\mathcal{X}$ (set of different outcomes) and probability mass function

$$p(x) = \mathbb{P}(X = x), \;\; x \in \mathcal{X}$$

The **entropy** $H(X)$ of X is

$$\begin{aligned} H(X) &= -\sum_{x\in\mathcal{X}} p(x)\,\log p(x) \\ &= \mathbb{E}\big(\log p(X)\big) = \mathbb{E}\Big(\log\frac{1}{p(X)}\Big) \end{aligned}$$

where the log is to the base 2 and entropy is expressed in bits. If the base of the logarithm is b, we denote the entropy as

$$H_b(X)$$

and if the base is e, the entropy is measured in **nats**. Clearly, $H(X) \geq 0$.

Note that entropy is a functional of the distribution of X. It does not depend on the actual values taken by the random variable X, but only on the probabilities.

Lemma 2.1.1. $H_b(X) = (\log_b a)\, H_a(X)$

Proof. $\log_b p = \log_b a \, \log_a p$ ■

Example 8. Let X be a discrete distribution with

$$P(X = a) = \frac{1}{2}, \;\; P(X = b) = \frac{1}{4}, \;\; P(X = c) = \frac{1}{8}, \;\; P(X = d) = \frac{1}{8}$$

Then

$$H(X) = -\frac{1}{2}\log\frac{1}{2} - \frac{1}{4}\log\frac{1}{4} - \frac{1}{8}\log\frac{1}{8} - \frac{1}{8}\log\frac{1}{8} = \frac{7}{4} \text{ bits}$$

Here is a nice interpretation of entropy. Given a discrete probability distribution X, suppose that we wish to determine the value of X with the minimum number of binary operations. For example, an efficient first question would be "Is $X = a$?" This splits the probability in half. If the answer to the first question is no, the second question can be "Is X = b?" The third question can be "Is X = c?" The resulting expected number of binary questions required is 7/4.

2.1.1 Joint Entropy and Conditional Entropy

Definition 2.1.2. The **joint entropy** $H(X,Y)$ of a pair of discrete random variables (X,Y) with a joint distribution $p(x,y)$ is defined

$$\begin{aligned} H(X,Y) &= -\sum_{x\in\mathcal{X}}\sum_{x\in\mathcal{Y}} p(x,y)\,\log p(x,y) \\ &= -\mathbb{E}\big(\log p(X,Y)\big) \end{aligned}$$

Definition 2.1.3. If $(X,Y) \sim p(x,y)$, the **conditional entropy** $H(Y|X)$ is defined as

$$\begin{aligned} H(Y|X) &= \sum_{x\in\mathcal{X}} p(x)\,H(Y|X=x) \\ &= -\sum_{x\in\mathcal{X}} p(x) \sum_{y\in\mathcal{Y}} p(y|x)\,\log p(y|x) \\ &- \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\,\log p(y|x) \\ &= -\mathbb{E}\big(\log p(Y|X)\big) \end{aligned}$$

The naturalness of the definition of joint entropy and conditional entropy is exhibited by the fact that the entropy of a pair of random variables is the entropy of one plus the conditional entropy of the other.

Lemma 2.1.2. Conditioning reduces entropy. That is,

$$H(X|Y) \leq H(X)$$

with equality if and only if X and Y are independent. Indeed, the "uncertainty" of X would decrease if we had any knowledge about a (potentially nonindependent) distribution Y.

Theorem 2.1.3 (Chain Rule for Entropy).

$$H(X,Y) = H(X) + H(Y|X)$$

Proof. By linearity of expectation,

$$\begin{aligned} H(X) + H(Y|X) &= -\mathbb{E}\Big(\log \frac{1}{p(X)}\Big) - \mathbb{E}\big(\log p(Y|X)\big) \\ &= -\mathbb{E}\big(\log p(X) + \log p(Y|X)\big) \\ &= -\mathbb{E}\big(\log p(X)p(Y|X)\big) \\ &= -\mathbb{E}\big(\log p(X,Y)\big) = H(X,Y) \end{aligned}$$

■

Corollary 2.1.3.1.

$$H(X,Y|Z) = H(X|Z) + H(Y|X,Z)$$

Corollary 2.1.3.2 (Chain Rule for Entropy). Let $X_1, X_2, ..., X_n$ be drawn according to $p(x_1, x_2, ..., x_n)$. Then

$$H(X_1, X_2, ..., X_n) = \sum_{i=1}^{n} H(X_i|X_{i-1}, ..., X_1)$$

Proof. By repeated application of the theorem. ■

Example 9. Let (X,Y) have the following joint distribution:

	1	2	3	4
1	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{32}$
2	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{32}$	$\frac{1}{32}$
3	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$
4	$\frac{1}{4}$	0	0	0

The marginal distribution of X is $\left(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}\right)$ and the marginal distribution of Y is $\left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right)$, meaning that

$$H(X) = \frac{7}{4} \text{ bits, } H(Y) = 2 \text{ bits}$$

Also,

$$\begin{aligned} H(X|Y) &= \sum_{i=1}^{4} p(Y=i)\, H(X|Y=i) \\ &= \frac{1}{4}H\left(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}\right) + \frac{1}{4}H\left(\frac{1}{4}, \frac{1}{2}, \frac{1}{8}, \frac{1}{8}\right) \\ &\quad + \frac{1}{4}H\left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right) + \frac{1}{4}H(1,0,0,0) \\ &= \frac{1}{4}\cdot\frac{7}{4} + \frac{1}{4}\cdot\frac{7}{4} + \frac{1}{4}\cdot 2 + \frac{1}{4}\cdot 0 = \frac{11}{8} \text{ bits} \end{aligned}$$

Similarly, $H(Y|X) = \frac{13}{8}$ bits and $H(X,Y) = \frac{27}{8}$ bits.

Note that while $H(X|Y) \neq H(Y|X)$,

$$H(X) - H(X|Y) = H(Y) - H(Y|X)$$

2.1.2 Relative Entropy and Mutual Information

Informally, the relative entropy is a measure of the distance between two distributions. That is, the relative entropy $D(p||q)$ is a measure of the inefficiency of assuming that the distribution is q when the true distribution is p. For example, if we knew the true distribution p of the random variable, we could construct a code with average description length $H(p$. If, instead, we used the code for a distribution q, we would need $H(p)+D(p||q)$ bits on the average to describe the random variable.

Definition 2.1.4 (Kullback-Leibler Divergence)**.** The **relative entropy**, or **Kullback-Leibler distance**, between two probability mass functions $p(x)$ and $q(x)$ is defined as

$$\begin{aligned} D(p||q) &= \sum_{x \in \mathcal{X}} p(x) \log \frac{p(x)}{q(x)} \\ &= \mathbb{E}_p\bigg(\log \frac{p(X)}{q(X)} \bigg) \end{aligned}$$

In other words, it is the expectation of the logarithmic difference between the probabilities P and Q, where the expectation is taken using the probabilities P. It is a measure of how one probability distribution is different from a second, reference probability distribution. A relative entropy of 0 indicates that p and q are identical. It is useful to interpret this measure as a "distance" between two distributions, but it is not a formal metric because it is not symmetric

$$D(p||q) = \mathbb{E}_p\bigg(\log \frac{p(X)}{q(X)} \bigg) \neq \mathbb{E}_q\bigg(\log \frac{q(X)}{p(X)} \bigg) = D(q||p)$$

and does not satisfy the triangle inequality.

Definition 2.1.5. Consider two random variables X and Y with a joint probability mass function $p(x, y)$ and marginal probability mass functions $p(x)$ and $p(y)$. The **mutual information** $I(X;Y)$ is the relative entropy between the joint distribution and product distribution $p(x)p(y)$.

$$\begin{aligned} I(X;Y) &= \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log \frac{p(x, y)}{p(x)p(y)} \\ &= D\big(p(x, y)||p(x)p(y)\big) \\ &= \mathbb{E}_{p(x,y)}\bigg(\log \frac{p(X, Y)}{p(X)p(Y)} \bigg) \end{aligned}$$

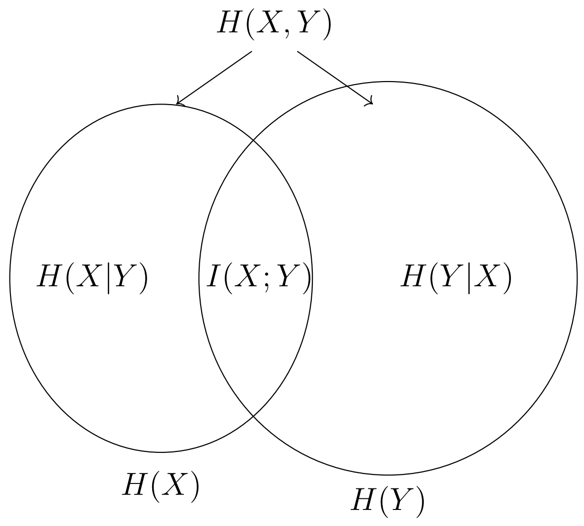
Theorem 2.1.4 (Mutual Information and Entropy). The mutual information $I(X;Y)$ is the reduction in the uncertainty of X due to the knowledge of Y.

$$I(X;Y) = H(X) - H(X|Y)$$

It follows that

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= H(Y) - H(Y|X) \\ &= H(X) + H(Y) - H(X, Y) \\ I(X;Y) &= I(Y;X) \\ I(X;X) &= H(X) \end{aligned}$$

We can visualize it as such:



Proof. We can write

$$\begin{aligned} I(X;Y) &= \sum_{x,y} p(x,y)\,\log\frac{p(x,y)}{p(x)p(y)} \\ &= \sum_{x,y} p(x,y)\log\frac{p(x|y)}{p(x)} \\ &= -\sum_{x,y} p(x,y)\log p(x) + \sum_{x,y} p(x,y)\log p(x|y) \\ &= -\sum_{x} p(x)\log p(x) - \Bigg(-\sum_{x,y} p(x,y)\log p(x|y)\Bigg) \\ &= H(X) - H(X|Y) \end{aligned}$$

By symmetry, we have

$$I(X;Y) = H(Y) - H(Y|X)$$

Thus, X says as much about Y as Y says about X. Since $H(X,Y) = H(X) + H(Y|X)$, we have

$$I(X;Y) = H(X) + H(Y) - H(X,Y) \implies I(X;X) = H(X) - H(X|X) = H(X)$$

That is, the mutual information of a random variable with itself is the entropy of the random variable. This is the reason that entropy is sometimes referred to as *self-information.* ■

Example 10. For the joint distribution in the previous example, the mutual information is

$$I(X;Y) = H(Y) - H(Y|X) = 2 - \frac{13}{8} = \frac{3}{8} \text{ bits}$$

Definition 2.1.6. The **conditional mutual information** of random variables X, Y, Z is the reduction in the uncertainty of X due to knowledge of Y when Z is given.

$$\begin{aligned} I(X;Y|Z) &= H(X|Z) - H(X|Y,Z) \\ &= E_{p(x,y,z)}\Bigg(\log\frac{p(X,Y|Z)}{p(X|Z)p(Y|Z)}\Bigg) \end{aligned}$$

Mutual information also satisfies a chain rule.

Theorem 2.1.5 (Chain rule for information).

$$I(X_1, X_2, ..., X_n; Y) = \sum_{i=1}^{n} I(X_i; Y|X_{i-1}, ..., X_1)$$

Proof.

$$\begin{aligned} I(X_1, ..., X_n; Y) &= H(X_1, ..., X_n) - H(X_1, ..., X_n|Y) \\ &= \sum_{i=1}^{n} H(X_i|X_{i-1}, ..., X_1) - \sum_{i=1}^{n} H(X_i|X_{i-1}, ..., X_1; Y) \\ &= \sum_{i=1}^{n} I(X_i; Y|X_1, X_2, ..., X_{i-1}) \end{aligned}$$

■

We now define a conditional version of relative entropy.

Definition 2.1.7. For joint probability mass functions $p(x, y)$ and $q(x, y)$, the **conditional relative entropy** $D\big(p(y|x)||q(y|x)\big)$ is the average of the relative entropies between the conditional probability mass functions $p(y|x)$ and $q(y|x)$ averaged over the probability mass function $p(x)$. That is,

$$\begin{aligned} D\big(p(y|x)||q(y|x)\big) &= \sum_{x} p(x) \sum_{y} p(y|x) \log \frac{p(y|x)}{q(y|x)} \\ &= \mathbb{E}_{p(x,y)}\left(\log \frac{p(Y|X)}{q(Y|X)} \right) \end{aligned}$$

2.1.3 Information Content

The **information content, self-information, surprisal**, or **Shannon information**, is a basic quantity derived from the probability of a particular event occurring from a random variable. It can be thought of as an alternative way of expressing probability, much like odds or log-odds, but which has particular mathematical advantages in the setting of information theory. The self-information can be interpreted as quantifying the level of "surprise" of a particular outcome. The information content can be expressed in various units of information, of which the most common is the "bit" (sometimes also called the *shannon*), as explained below.

The definition of self-information was chosen to meet several axioms:

1. An event with probability 100% is perfectly unsurprising and yields no information.
2. The less probable an event is, the more surprising it is and the more information it yields.
3. If two independent events are measured separately, the total amount of information is the sum of the self-informations of the individual events.

It can be shown that there is a unique function of probability that meets these three axioms, up to a multiplicative scaling factor. Broadly given an event x with probability P, the information content is defined as follows:

$$I(x) \equiv -\log_b\big(\mathbb{P}(x)\big)$$

The base of the log is left unspecified, which corresponds to the scaling factor above. Formally, given a continuous random variable X with probability density function $p_X(x)$, the self-information of measuring X as outcome x is defined as:

$$I_X(x) \equiv -\log\big(p_X(x)\big) = \log\left(\frac{1}{p_X(x)}\right)$$

Properties

For a given probability space, the measurement of rarer events are intuitively more "surprising," and yield more information content, than more common values. Thus, self-information is a strictly decreasing monotonic function of the probability. While standard probabilities are represented by real numbers in the interval $[0,1]$, self-informations are represented by extended real numbers in the interval $[0,\infty]$. In particular, we have the following, for any choice of logarithmic base:

1. If a particular event has a 100% probability of occurring, then its self-information is $-\log(1) = 0$: its occurrence is *perfectly non-surprising* and yields no information.
2. If a particular event has a 0% probability of occurring, then its self-information is $-\log(0) = \infty$: its occurrence is *infinitely surprising.*

From this, we can get a few general properties:

1. Intuitively, more information is gained from observing an unexpected event—it is *surprising.*
2. This establishes an implicit relationship between the self-information of a random variable and its variance.

Note also that this definition of information content satisfies additivity. Consider two independent random variables X, Y with probability mass functions $p_X(x)$ and $p_Y(y)$ respectively. The joint probability mass function is

$$p_{X,Y}(x,y) \equiv \mathbb{P}(X = x, Y = y) = p_X(x)\,p_Y(y)$$

because X and Y are independent. The information content of the outcome $(X,Y) = (x,y)$ is

$$\begin{aligned} I_{X,Y}(x,y) &= -\log_2\big(p_{X,Y}(x,y)\big)] \\ &= -\log_2\big(p_X(x)\,p_Y(y)\big) \\ &= -\log_2\big(p_X(x)\big) - \log_2\big(p_Y(y)\big) \\ &= I_X(x) + I_Y(y) \end{aligned}$$

A fair coin toss, which can be measured by the Bernoulli distribution $\mathbb{P}(H) = \frac{1}{2},\ \ \mathbb{P}(T) = \frac{1}{2}$ has the information contents (in bits, base 2)

$$I_X(H) = -\log_2\big(\mathbb{P}(X = H)\big) = -\log_2\frac{1}{2} = 1 I_X(T) \quad = -\log_2\big(\mathbb{P}(X = T)\big) = -\log_2\frac{1}{2} = 1$$

A fair six-sided die roll has the discrete uniform distribution. The information content is

$$I_X(1) = I_X(2) = I_X(3) = I_X(4) = I_X(5) = I_X(6) = -\log_2 \frac{1}{6} \approx 2.585$$

Two independent, identically distributed dice gives an information content of

$$I_{X,Y}(x, y) = -\log_2 \frac{1}{36} \approx 5.169925$$

where $1 \leq x, y \leq 6$. Note that this could have also been calculated by simply adding the self-information of one die with that of another identical die.

2.1.4 Entropy

The **entropy** of a random variable is the average level of *information*, *surprise*, or *uncertainty* inherent in the variable's possible outcomes. As an example, consider a biased coin with probability p of landing on heads and probability 1-p of landing on tails. The maximum surprise is for p = 1/2, when there is no reason to expect one outcome over another, and in this case a coin flip has an entropy of one bit. The minimum surprise is when p = 0 or p = 1, when the event is known and the entropy is zero bits. Other values of p give different entropies between zero and one bits.

Given a discrete random variable X, with possible outcomes $x_1, x_2, ..., x_n$ which occur with probability $P(x_1), P(x_2), ..., P(x_n)$, the entropy of X is formally defined as:

$$\begin{aligned} H(X) &\equiv -\sum_{i=1}^{n} \mathbb{P}(x_i) \log \mathbb{P}(x_i) \\ &\equiv \mathbb{E}\big(I_X(X)\big) \end{aligned}$$

which is the expected information content of measurement of X. Base 2 gives the unit of bits, while base e gives the *natural units* nat, and base 10 gives a unit called dits.

The entropy was originally created by Shannon as part of his theory of communication, in which a data communication system is composed of three elements: a source of data, a communication channel, and a receiver. In Shannon's theory, the "fundamental problem of communication" – as expressed by Shannon – is for the receiver to be able to identify what data was generated by the source, based on the signal it receives through the channel. Shannon considered various ways to encode, compress, and transmit messages from a data source, and proved in his famous source coding theorem that the entropy represents an absolute mathematical limit on how well data from the source can be losslessly compressed onto a perfectly noiseless channel.

The English text, treated as a string of character, has fairly low entropy, i.e. is fairly predictable. If we do not know exactly what is going to come next, we can be fairly certain that, for example, 'e' will be far more common than 'z', that the combination 'qu' will be much more common than any other combination with a 'q' in it, and that the combination 'th' will be more common than 'z', 'q', or 'qu'. After the first few letters one can often guess the rest of the word. English text has between 0.6 and 1.3 bits of entropy per character of the message.

Shannon's Source Coding Theorem

If a compression scheme is lossless – one in which you can always recover the entire original message by decompression – then a compressed message has the same quantity of information as the original, but communicated in fewer characters. It has more information (higher entropy) per character. A compressed message has less redundancy. Shannon's source coding theorem states a lossless compression scheme cannot compress messages, on average, to have more than one bit of information per bit of message, but that any value less than one bit of information per bit of message can be attained by employing a suitable coding scheme. The entropy of a message per bit multiplied by the length of that message is a measure of how much total information the message contains.

This theorem establishes the limits to possible data compression. Informally, it states that

N i.i.d. random variables each with entropy $H(X)$ can be compressed into more than N $H(X)$ bits with negligible risk of information loss, as $N \to \infty$; but conversely, if they are compressed into fewer than N $H(X)$ bits, it is virtually certain that information will be lost.

2.2 Data Compression

Data compression, also called **source coding** or **bit-rate reduction**, is the process of encoding information using fewer bits than the original representation. Data compression algorithms can be categorzied into two types:

1. **Lossless compression** reduces bits by identifying and eliminating statistical redundancy. No information is lost in lossless compression.
2. **Lossy compression** reduces bits by removing unnecessary or less important information.

Typically, a device that performs data compression is referred to as an **encoder**, and one that performs the reversal of the process (decompression) as a **decoder**.

A **space–time** or **time–memory trade-off** in computer science is a case where an algorithm or program trades increased space usage with decreased time. Here, space refers to the data storage consumed in performing a given task (RAM, HDD, etc), and time refers to the time consumed in performing a given task (computation time or response time).

A space–time trade-off can be applied to the problem of data storage. If data is stored uncompressed, it takes more space but access takes less time than if the data were stored compressed (since compressing the data reduces the amount of space it takes, but it takes time to run the decompression algorithm). Depending on the particular instance of the problem, either way is practical. There are also rare instances where it is possible to directly work (which may also be faster) with compressed data.

Data Compression Ratio

The **data compression ratio**, also known as the **compression power**, is a measurement of the relative reduction in size of data representation produced by a data compression algorithm. It is defined as

$$\text{Compression Ratio} = \frac{\text{Uncompressed Size}}{\text{Compressed Size}}$$

For example, a representation that compresses a file's storage size from 10MB to 2MB has a compression ratio of $10/2 = 5$. We can alternatively talk about the **space saving**, which is defined

$$\text{Space Saving} = 1 - \frac{\text{Compressed Size}}{\text{Uncompressed Size}}$$

So, the previous representation yields a space saving of 0.8, or 80%.

Lossless compression of digitized data such as video, digitized film, and audio preserves all the information, but it does not generally achieve compression ratio much better than 2:1 because of the **intrinsic entropy** of the data. Compression algorithms which provide higher ratios either incur very large overheads or work only for specific data sequences (e.g. compressing a file with mostly zeros). In contrast, lossy compression (e.g. JPEG for images, or MP3 and Opus for audio) can achieve much higher compression ratios at the cost of a decrease in quality, such as Bluetooth audio streaming, as visual or audio compression artifacts from loss of important information are introduced. In general, whether a compression ratio is high or not really depends on what kind of data is being compressed and how it is compressed.

2.2.1 Lossless Compression

Lossless data compression algorithms usually exploit statistical redundancy to represent data without losing any information, so that the process is reversible. Lossless compression is possible because most real-world data exhibits statistical redundancy. For example, an image may have areas of color that do not change over several pixels; instead of coding "red pixel, red pixel, ..." the data may be encoded as "279 red pixels". This is a basic example of run-length encoding; there are many schemes to reduce file size by eliminating redundancy.

A **dictionary coder**, also known as a **substitution coder**, is a class of lossless data compression algorithms which operate by searching for matches between the text to be compressed and a set of strings contained in a data structure (called the 'dictionary') maintained by the encoder. When the encoder finds such a match, it substitutes a reference to the string's position in the data structure.

Some dictionary coders use a *static dictionary*, one whose full set of strings is determined before coding begins and does not change during the coding process. This approach is most often used when the message or set of messages to be encoded is fixed and large; for instance, an application that stores the contents of a book in the limited storage space of a PDA generally builds a static dictionary from a concordance of the text and then uses that dictionary to compress the verses.

In a related and more general method, a dictionary is built from redundancy extracted from a data environment (various input streams) which dictionary is then used statically to compress a further input stream. For example, a dictionary is built from old English texts then is used to compress a book. More common are methods where the dictionary starts in some predetermined state but the contents change during the encoding process, based on the data that has already been encoded.

Run-length Encoding (RLE) Compression

RLE is a form of lossless data compression in which *runs* of data (sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. This is most useful on data that contains many such runs.

For example, consider a screen containing plain black text on a solid white background. There will be many long runs of white pixels in the blank space, and many short runs of black pixels within the text. A hypothetical scan line, with B representing a black pixel and W representing white, might read as follows:

```
WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWW
```

With a RLE data compression algorithm applied to the above hypothetical scan line, it can be rendered as follows:

```
12W1B12W3B24W1B14W
```

which can be interpreted as a sequence of 12 Ws, 1 B, 12 Ws, 3 Bs, and so on. This run-length code represents the original 67 characters in only 18. While the actual format used for the storage of images is generally binary rather than ASCII characters like this, the principle remains the same. Even binary data files can be compressed with this method.

Lempel-Ziv (LZ) Compression

The **LZ77** and **LZ78** (also known as the **LZ1** and **LZ2**), respectively, are lossless data compression algorithms published by Lempel and Ziv in 1977/78. They obsolete themselves but form the basis for many modern variations including **LZW, LZSS, LZMA**, and others.

LZ77 and 78 are both dictionary coders, but are not static. Rather, the dictionary starts in some predetermined state but the contents change during the encoding process.

DEFLATE Compression

2.2.2 Huffman Coding

2.2.3 Lossy Compression

Most forms of lossy compression are based on **transform coding**, such as the **discrete cosine transform (DCT)**. Another type of compression is the **singular value decomposition (SVD)**.

Chapter 3

Finite Computation

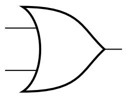
3.1 AON-CIRC and Straight Line Programs

Definition 3.1.1. 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

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

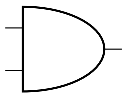
   An **OR gate** has two incoming wires and one (or more) outgoing wires.



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

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

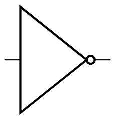
   An **AND gate** has two incoming wires and one (or more) outgoing wires.



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 11. 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 3.1.2. 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

$x =$ `(X[0], X[1], ..., X[n-1])`

and output variables of the form

$y =$ `(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

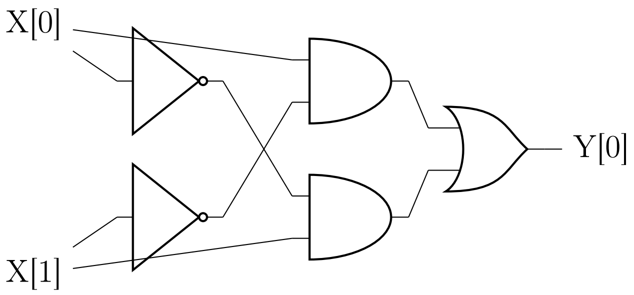
$$f : \{0,1\}^n \longrightarrow \{0,1\}^m$$

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

Example 12. 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:



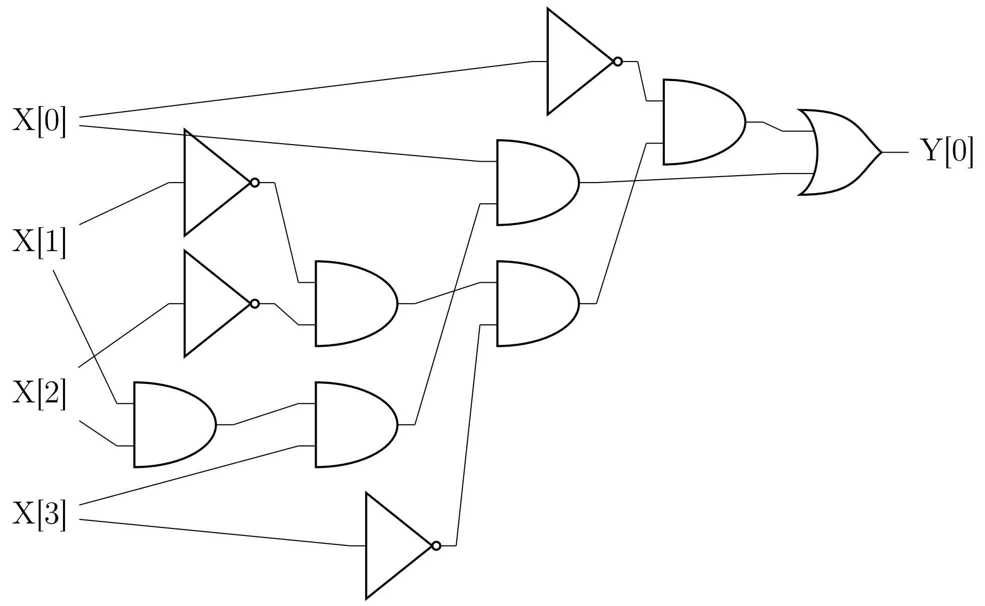
This can be computed with the straight-line algorithm as such. Given (a, b) as inputs, we have $w_1 = AND(a, b), w_2 = NOT(w_1)$, and $w_3 = OR(a, b)$. Then the algorithm returns $AND(w_2, w_3)$. In Python, this can be programmed:

```python
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 3.1.1 (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 13. 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:



Topological Sortings of Graphs

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

Definition 3.1.3 (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 3.1.4 (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 3.1.5 (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 3.1.2. 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 3.1.2.1. There exists a layering for every directed acyclic graph. That is, every DAG can be topologically sorted.

Formal Definition of Boolean Circuits

Definition 3.1.6. 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.

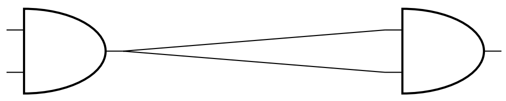
3. Exactly m of the gates are also labeled with the m labels

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

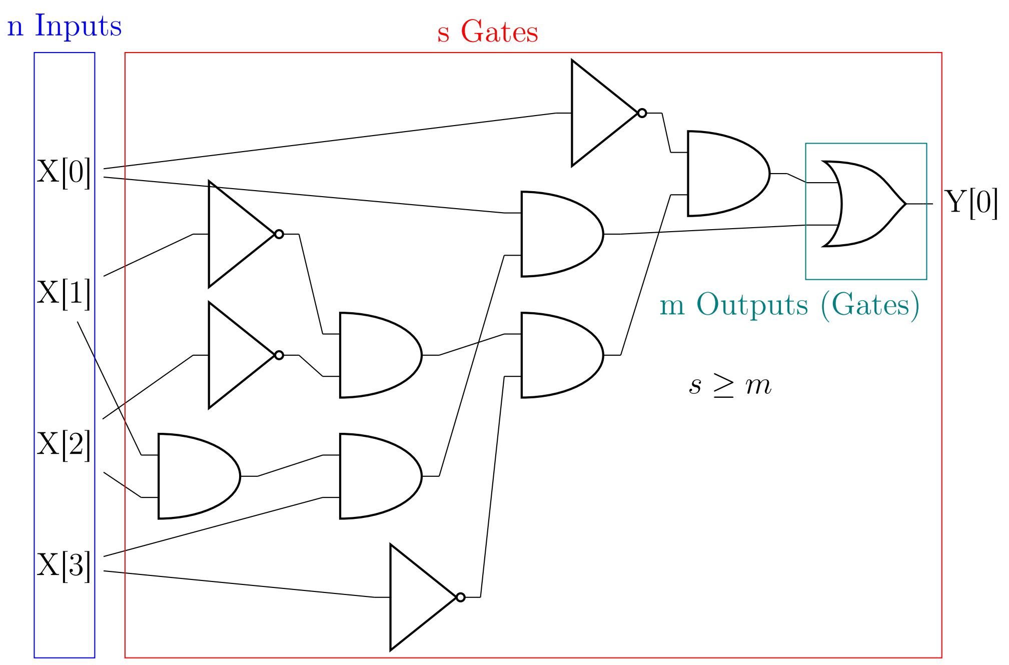
   in addition to their label $\wedge/\vee/\neg$. These are known as **outputs**.

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 3.1.7. 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.

3.1.1 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.

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.

3.1.2 The NAND Function

Definition 3.1.8. 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 3.1.3 (Universality of NAND). We can compute AND, OR, and NOT by composing only the NAND function.

Proof. We can see that, using double negation,

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

■

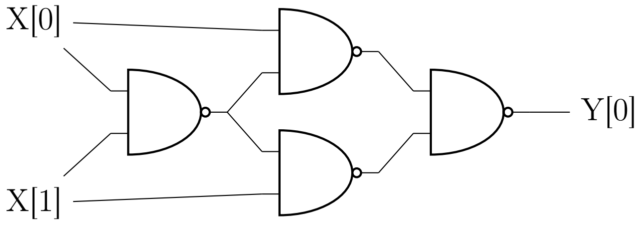
Corollary 3.1.3.1. For every Boolean circuit C of s gates, there exists a NAND circuit C' of at most $3s$ gates that computes the same function as C.

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

NAND Circuits

Definition 3.1.9. A **NAND Circuit** is a circuit in which all the gates are NAND operations. Despite their simplicity, NAND circuits can be quite powerful.

Example 14. 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 3.1.10. 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 3.1.4 (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.

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 3.1.11 (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 15. Let $\mathcal{F} = \{IF, ZERO, ONE\}$ where

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

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.

3.1.3 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 16. 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 3.1.12. 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.

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 3.1.13. 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 3.1.14. 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 3.1.5. 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.

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 3.1.6 (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 3.1.7 (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 $MULT_n$.

The Lookup Function

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

Definition 3.1.15 (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 3.1.8. 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.

Computing Every Function

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

Theorem 3.1.9 (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 3.1.16. 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:

$$\begin{tikzcd}
\text{AON-Bool Circuit} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & \text{NAND-Bool Circuit} \arrow[d, leftrightarrow] \\
\text{AON-CIRC Program} \arrow[r, leftrightarrow] & \text{NAND-CIRC Program} \arrow[d, leftrightarrow] \arrow[dl, leftrightarrow, "\text{Syntactic Sugar}"'] \\
\text{NAND-CIRC-PROC} & \text{NAND-CIRC-IF}
\end{tikzcd}$$

3.2 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**.

3.2.1 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 3.2.1 (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 $cs \log s$.

3.2.2 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 3.2.2 (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 3.2.3 (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**.

3.2.3 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 3.2.1 (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 17. 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

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

3.2.4 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 3.2.4. 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 3.2.2. 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 3.2.5 (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.

Chapter 4

Uniform Computation

4.1 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.

4.1.1 Functions with Inputs of Unbounded Length

Example 18. 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 19. 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 20 (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 4.1.1. Sometimes, we can obtain a Boolean variant of a non-Boolean function. This process is called **booleanizing**.

Example 21 (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.

4.1.2 Deterministic Finite Automata

Definition 4.1.2. 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**.

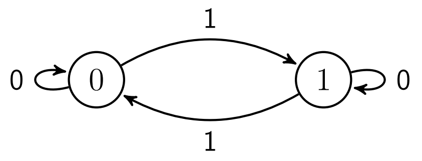
We can think of such an algorithm as a "machine" that can be in one of C states, for some constant C. The machine starts in some initial state and then reads its input $x \in \{0, 1\}^*$ one bit at a time. Whenever the machine reads a bit $\sigma \in 0, 1$, it transitions into a new state based on σ 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 4.1.3. 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 4.1.4. We say that a function $F : \{0, 1\}^* \longrightarrow \{0, 1\}$ is **DFA computable** if there exists some DFA that computes F.

Theorem 4.1.1. 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 4.1.2. The set of all Boolean functions $\{f \mid f : \mathbb{N} \longrightarrow \{0, 1\}\}$ are uncountable.

Corollary 4.1.2.1 (Existence of DFA-uncomputable functions). There exists a Boolean function $F : \{0, 1\}^* \longrightarrow \{0, 1\}$ that is not computable by *any* DFA.

4.1.3 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 4.1.5. 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 22. 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 4.1.6. 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 4.1.7. 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 4.1.8. The set of functions computable by DFAs is the same as the set of languages that can be recognized by regular expressions.

4.2 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 4.2.1 (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:

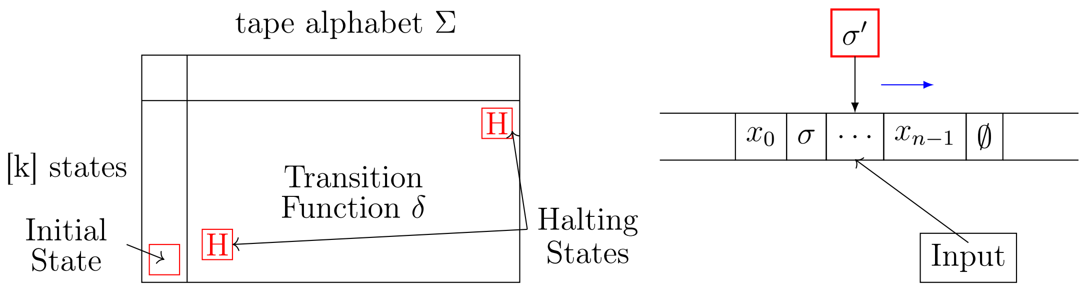
1. We initialize T to be the infinite sequence (also represented by a tape)
$$\triangleright, x_0, x_1, ..., x_{n-1}, \emptyset, \emptyset, ...$$
where $n = |x|$. That is, $T[0] = \triangleright, T[i+1] = x_i$ for $i \in [n]$, and $T[i] = \emptyset$ for $i > n$.)
2. We also initialize $i = 0$ (the head 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 23 (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 4.2.2 (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 4.2.3 (The class **R**)**.** We define **R** to be the set of all computable functions $F : \{0,1\}^* \longrightarrow \{0,1\}$.

4.2.1 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 4.2.4 (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 4.2.1 (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.

Uniformity of Computation

Definition 4.2.5. 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.

4.2.2 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 4.2.6. 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 4.2.2. 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.

4.3 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 4.3.1. 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 4.3.2 (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 $\mathcal{M}(P)$.

4.3.1 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 4.3.3. 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 4.3.1. Conway's Game of Life is Turing complete.

One-Dimensional Cellular Automata

Definition 4.3.4. 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 4.3.2. The Rule 110 cellular automaton is Turing complete. That is, any calculation or computer program can be simulated using this automaton.

Definition 4.3.5 (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.

4.3.2 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 4.3.6 (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

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

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 24. 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.
$$\mathsf{add} : a \mapsto \mathsf{adda}$$
2. `adda` takes argument b and adds b to the predetermined number a.
$$\mathsf{adda} : b \mapsto a + b$$

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 $(x, y) \mapsto x + y$.

Abstractions

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

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

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 $\lambda\ a.\lambda\ b.a = \lambda\ a.(\lambda\ b.a)$.

Beta Reduction

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

Example 25. We can β reduce the expression into its simplest form, called the **beta normal form**.

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

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

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 4.3.7. 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 26. The following λ expression can be simplified as such:

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

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:

$$\begin{tikzcd}
\text{1,2-dim. cellular automata} \arrow[dr, leftrightarrow] & & & \text{General Purpose programs} \arrow[dl, leftrightarrow, "\text{Syntactic Sugar}"] \\
& \text{Turing Machines} \arrow[r, leftrightarrow] \arrow[dd, leftrightarrow] & \text{NAND-TM Programs} \arrow[dd, leftrightarrow] & \\
\lambda\text{-calculus} \arrow[ur, leftrightarrow] & & & \\
& \text{RAM Machines} \arrow[r, leftrightarrow] & \text{NAND-RAM Programs} &
\end{tikzcd}$$

4.4 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 4.4.1 (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 4.4.1 (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.

4.4.1 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 4.4.2. 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 4.4.2. 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.

4.4.2 Impossibility of General Software Verification

Definition 4.4.3. 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 27. 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 28. Consider the following two C programs:

```
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 4.4.4 (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 4.4.3 (Rice's Theorem). Let $F : \{0,1\}^* \longrightarrow \{0,1\}$. If F is semantic and non-trivial, then it is uncomputable.

Corollary 4.4.3.1. 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 $\{0,1\}^* \longrightarrow \{0,1\}$.

4.4.3 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 4.4.5 (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 29. 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*.

Chapter 5

Efficient Algorithms

5.1 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.

5.1.1 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).

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.

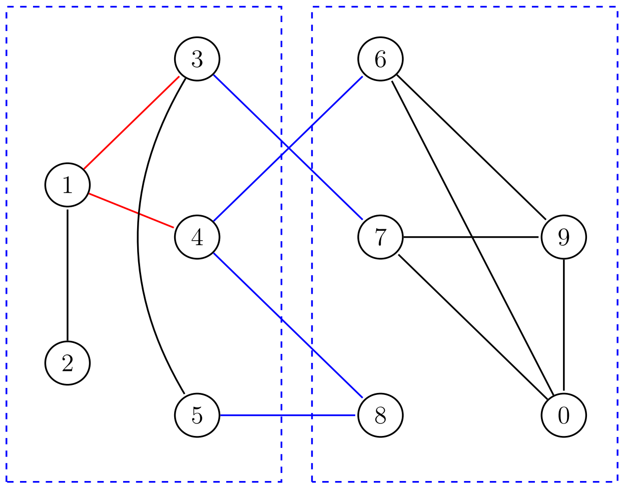
Surprisingly, while we can find the shortest path in $O(m)$ time, there is no known algorithm for the longest path problem that significantly improves on the trivial "exhaustive search" or "brute force" algorithm that enumerates all the exponentially many possibilities for such paths. Specifically, the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

Finding the minimum cut in a graph

Definition 5.1.1. 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.

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 5.1.1 (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.

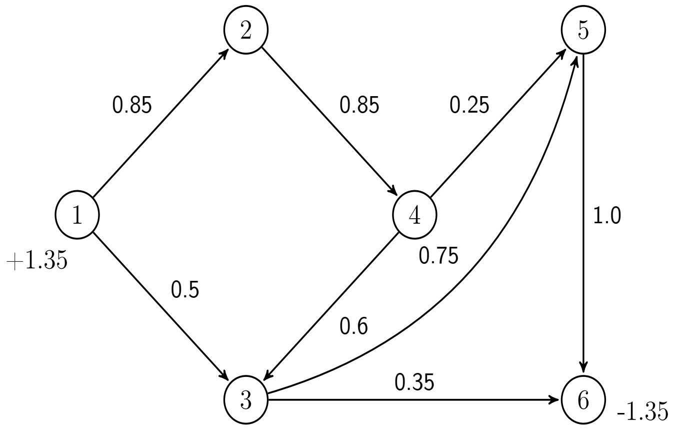
It is easy to see why this theorem is when we interpret the minimum cut S acting as a bottleneck that restricts the flow the most. The Max-Flow Min-Cut Theorem reduces the task of computing a minimum cut of the task of computing a *maximum flow.* For this problem, the *Ford-Fulkerson Algorithm* is direct 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 5.1.2. 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:

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

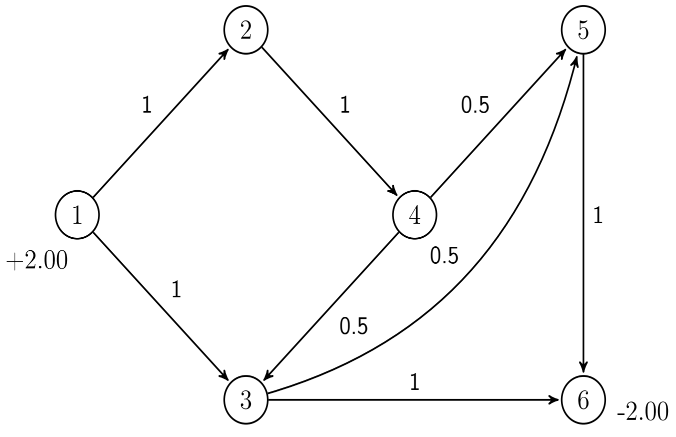
We write the source and sinks as summations since there may be multiple sources and sinks.

Example 30. 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 5.1.3. 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 5.1.2. 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.

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 5.1.3. 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.

5.1.2 Computational Problems Beyond Graphs

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 5.1.4. 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 31. 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 5.1.5. 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 32. 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.

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

$$a_{i,j}x_jx_k$$

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.

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 5.1.6. 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, 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.

Finding a Zero-Sum Equilibrium

Definition 5.1.7. 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.

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.

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.

5.1.3 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.

5.2 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).

5.2.1 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 5.2.1. 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.

Polynomial and Exponential Time

Definition 5.2.2. 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
$$\mathbf{P} = \bigcup_{c \in \{1,\ldots m\}} TIME_{\mathsf{TM}}\big(n^c\big), \quad m \in \mathbb{N}$$
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
$$\mathbf{EXP} = \bigcup_{c \in \{1,\ldots,m\}} TIME_{\mathsf{TM}}\big(2^{n^c}\big)$$
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 5.2.1. 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.

5.2.2 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 5.2.3. 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 $F(x)$.

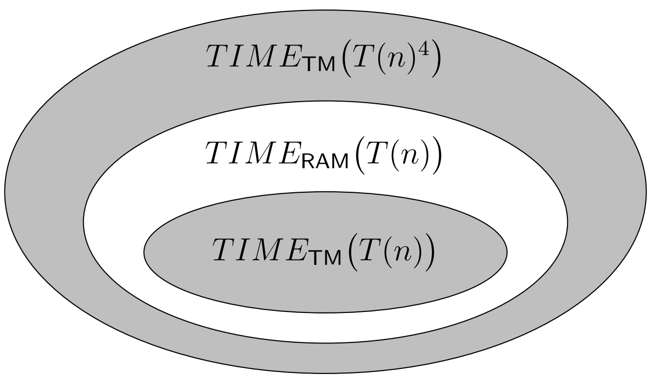
We define $TIME_{\mathsf{RAM}}\big(T(n)\big)$ to be the set of Boolean functions ($\{0,1\}^* \longrightarrow \{0,1\}$) that are computable in $T(n)$ RAM time.

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 5.2.2 (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


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

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 5.2.4. 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.

5.2.3 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 5.2.3 (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.

5.2.4 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 5.2.4 (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.

5.2.5 Non-Uniform Computation

5.3 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**.

5.3.1 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 5.3.1 (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.

5.3.2 Reducing 3SAT to Zero-One and Quadratic Equations

Definition 5.3.2. 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

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

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

$$\begin{aligned} x_0 + x_1 + x_2 &= 2 \\ x_0 + x_2 &= 1 \\ x_1 + x_2 &= 2 \end{aligned}$$

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 5.3.1 (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:

$$\begin{aligned} x_0^2 - x_0 &= 0 \\ x_1^2 - x_1 &= 0 \\ x_2^2 - x_2 &= 0 \\ 1 - x_0 - x_1 + x_0 x_1 &= 0 \end{aligned}$$

Theorem 5.3.2 (Hardness of Quadratic Equations).

$$3SAT \leq_p QUADEQ$$

5.3.3 Independent Set and Other Graph Problems

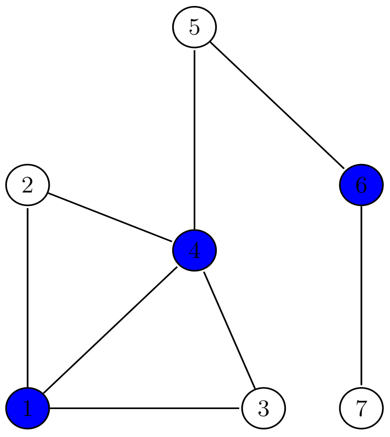
Definition 5.3.3. For a graph $G = (V, E)$, an **independent set**, also known as a **stable set**, is a subset $S \subseteq V$ such that there are no edges with both endpoints in S (in other words, $E(S, S) = \emptyset$). Trivially, every singleton (of one point) is an independent set.

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 5.3.3 (Hardness of Independent Set).

$$3SAT \leq_p ISET$$

Definition 5.3.4. 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 5.3.4.

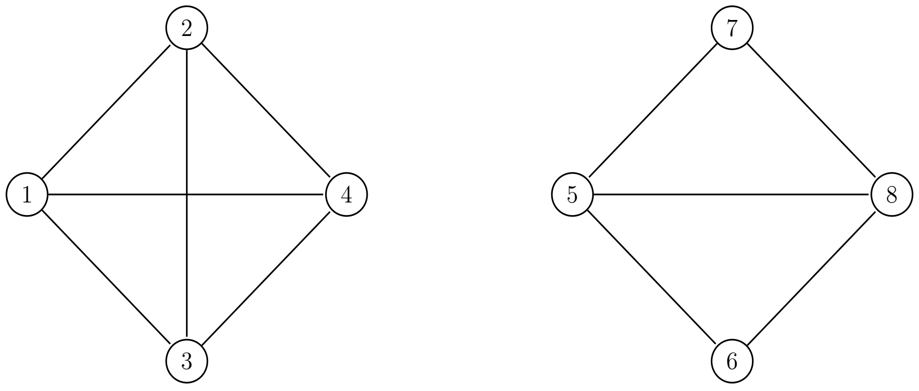
$$3SAT \leq_p VC$$

Definition 5.3.5. 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 5.3.5.

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

Definition 5.3.6. 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

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 5.3.6 (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.

5.4 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.

5.4.1 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 5.4.1 (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.

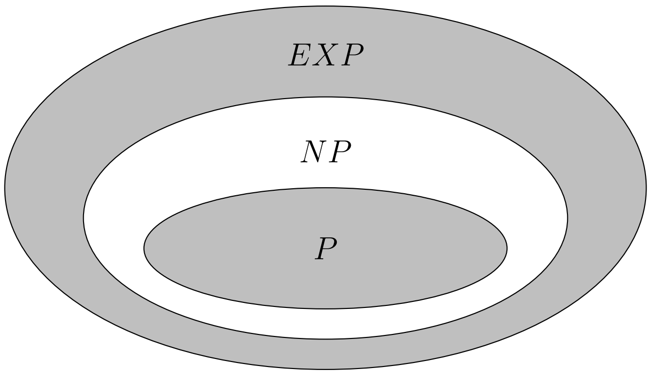
5. $MAXCUT \in \mathbf{NP}$ since for every graph G and integer k, $MAXCUT(G,k) = 1$ if and only if there exists a cut $(S, \overline{S})$ in G that cuts at least k edges. We can check that condition that $(S, \overline{S})$ is a cut of value at least k in polynomial time by checking that S is a subset 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 5.4.1. 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. ∎

5.4.2 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 5.4.2 (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 5.4.2. 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 $\boldsymbol{P} = \boldsymbol{NP}$.*

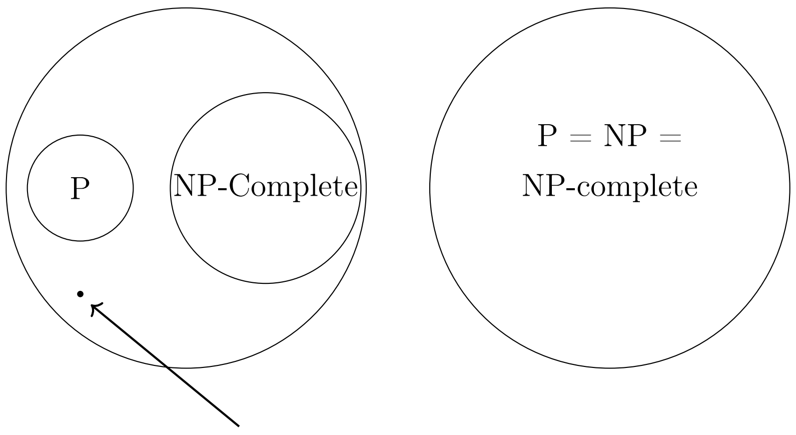
5.4.3 P = NP?

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 **P** is a strict subset of **NP** and by Lander's theorem, **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)



problem that is neither P nor NP complete (by Lander's Theorem)

5.4.4 NANDSAT, 3NAND Problems

Definition 5.4.3. 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 5.4.4. 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 33. 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 5.4.3. *$NANDSAT$ and $3NAND$ is* **NP** *complete.*

Chapter 6

Randomized Computation

6.1 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*.

6.1.1 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 6.1.1 (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.

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 6.1.2. 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

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

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 $m/2$. ■

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

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

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

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

This can be generalized in the following lemma.

Lemma 6.1.3. 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. ■

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

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

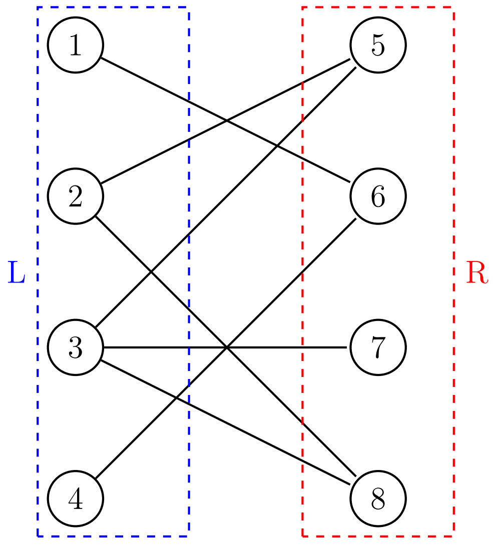
$$ST = \tilde{O}\big((4/3)^n\big) = \tilde{O}(1.\overline{3}^n)$$

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

Theorem 6.1.5 (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 $\frac{1}{2}$.

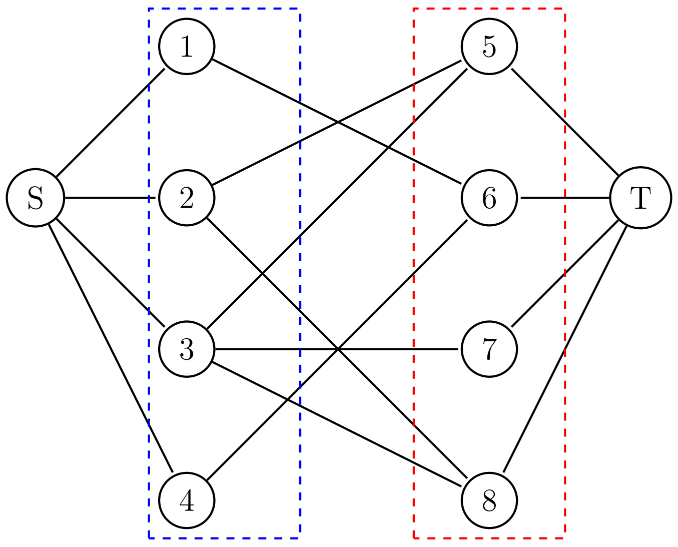
Bipartite Matching

Definition 6.1.1. 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 6.1.6 (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 6.1.7 (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

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

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.

6.2 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.

6.2.1 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 6.2.1 (The class **BPP**)**.** Let $F : \{0, 1\}^* \longrightarrow \{0, 1\}$. We say that $F \in$ **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 6.2.2 (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 **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 6.2.1 (Sipser-Gacs Theorem)*.* If $\mathbf{P} = \mathbf{NP}$ then $\mathbf{BPP} = \mathbf{P}$.

Success Amplification of two-sided error algorithms

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

Theorem 6.2.2 (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 $x \in \{0,1\}^n$,

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

BPP and NP Completeness

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

Theorem 6.2.3*.* 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**.

6.2.2 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 6.2.4 (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,

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

Chapter 7

Machine Learning

Stanford's CS 229 Lectures by Andrew Ng.

7.1 Supervised Learning

We begin by establishing some notation. The input variables are usually denoted with the letter x (which lies in the input space $\mathcal{X}$ and the outputs with y (lying in the output space $\mathcal{Y}$). For example, say that we have the dataset:

x_1	...	x_d	y
$x_1^{(1)}$	...	$x_d^{(1)}$	$y^{(1)}$
$x_1^{(2)}$	...	$x_d^{(2)}$	$y^{(2)}$
...	...	...	...
$x_1^{(n)}$	...	$x_d^{(n)}$	$y^{(n)}$

Note that:

1. d represents the number of parameters, represented by a subscript.
2. n represents the number of training samples, represented by a superscript.
3. $(x^{(i)}, y^{(i)})$ is called a **training example**.

To describe the supervised learning problem, our goal is, given a training set, to learn the **hypothesis function** $h : \mathcal{X} \longrightarrow \mathcal{Y}$ that is a good predictor for the corresponding value of y.

7.1.1 Linear Regression

It is safe to assume that in most cases, $\mathcal{X} = \mathbb{R}^n$ and $\mathcal{Y} = \mathbb{R}$ (or a subset of it). Our goal is to find an (affine) linear hypothesis function h of the form

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n$$

We can drop the subscript θ from h_θ if there is no risk of confusion to get (assuming $x_0 = 1$)

$$h(x) = \sum_{i=0}^{d} \theta_i x_i = \theta^T x$$

To determine how well of an approximation h is to the actual values of y, we define the **cost function**

$$J(\theta) \equiv \frac{1}{2}\sum_{i=1}^{n}\big(h_\theta(x^{(i)}) - y^{(i)}\big)^2$$

Even though J takes in θ values, this is really equivalent to J taking in h since h is completely determined by θ. Note that $J(\theta)$ is a function itself that takes in input values θ_i, forms the linear function $h_\theta(x)$, and computes the sums of all the squares of its residuals from the n data points. That is,

$$J : \mathbb{R}^d \longrightarrow \mathbb{R}_0^+$$

is a smooth function (this smoothness criterion is important).

Batch Gradient Descent

We would like to minimize J, that is get it as close to 0 as possible (since $J \geq 0$). In the space $\mathbb{R}^d$ of θ_j's, we start off at an initial θ and go in the direction opposite of that of the gradient.

$$\theta = \theta - \alpha\,\nabla J(\theta) \iff \begin{cases} \theta_1 = \theta_1 - \alpha\frac{\partial}{\partial\theta_1}J(\theta) \\ \vdots \\ \theta_j = \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) \\ \vdots \\ \theta_d = \theta_d - \alpha\frac{\partial}{\partial\theta_d}J(\theta) \end{cases}$$

The α is a scalar constant called the **learning rate** (i.e. how large each step is going to be). We can explicitly evaluate $\nabla J(\theta)$ as

$$\begin{aligned} \frac{\partial}{\partial\theta_j} &= \frac{\partial}{\partial\theta_j}\bigg(\frac{1}{2}\sum_{i=1}^{n}\big(h_\theta(x^{(i)}) - y^{(i)}\big)\bigg) \\ &= \dots \\ &= \sum_{i=1}^{n}\big(h_\theta x^{(i)} - y\big)x_j^{(i)} \end{aligned}$$

Therefore, our gradient descent algorithm in vector form is (with python syntax):

```
while convergence not met:
    $\theta = \theta + \alpha\sum_{i=1}^{n}\big(y^{(i)} - h_\theta(x^{(i)})\big)x^{(i)}$
```

In coordinate notation, we have

```
while convergence not met:
    for j in 1,...,d:
        $\theta_j = \theta_j + \alpha\sum_{i=1}^{n}\big(y^{(i)} - h_\theta(x^{(i)})\big)x_j^{(i)}$
```

Note that:

1. This algorithm repeatedly takes a step in the direction of steepest decrease of J
2. The magnitude of the update is proportional to the **error** term $y^{(i)} - h_\theta(x^{(i)})$.
3. This algorithm can be susceptible to local minima.
4. The algorithm may not converge if α (the "step size") is too high.

Note that each update step in gradient descent requires solving all d coefficients of the coordinate vectors of $\nabla J(\theta)$, and within each coordinate calculation, we must *iterate through all n training samples*, for a total of $n \times d$ calculations. GD by going through all training samples is known as **batch gradient descent**.

Stochastic, Incremental Gradient Descent

Rather than updating the vector θ in batches, we can apply **stochastic GD** that works incrementally by updating θ with each term in the summation. In vector form,

```
while convergence not met:
    for i in 1,...,n:
        $\theta = \theta + \alpha\big(y^{(i)} - h_\theta(x^{(i)})\big)x^{(i)}$
```

and in coordinate form,

```
while convergence not met:
    for i in 1,...,n:
        for j in 1,...d
            $\theta_j = \theta_j + \alpha\big(y^{(i)} - h_\theta(x^{(i)})\big)x_j^{(i)}$
```

Whereas batch GD has to scan through the entire training set before taking a single step (a costly operation if n is large), **stochastic gradient descent** updates the parameters with respect to a single training example and can start making progress right away. Often, stochastic gradient descent has a much faster convergence rate (if it does converge).

7.1.2 The Normal Equations

Rather than using an iterative method, we can explicitly minimize the cost function. Let us review a bit of matrix derivatives. Let $A = (A)_{ij}$ be a $n \times d$ matrix and $f : \mathbb{R}^{n\times d} \longrightarrow \mathbb{R}$ be a real function over the space of matrices. Then, the derivative of f with respect to A is:

$$\nabla_A f(A) = \begin{pmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1d}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial A_{n1}} & \cdots & \frac{\partial f}{\partial A_{nd}} \end{pmatrix}$$

So, the gradient $\nabla_A f(A)$ is itself an $n \times d$ matrix, with each entry being the derivative in the direction of the canonical basis vector of $\mathbb{R}^{n\times d}$.

Least Squares Revisited

Let us define the **design matrix** X to be the $n \times d$ matrix that contains the training examples' input values in its rows. Additionally, let y be the n-dimensional vector containing all the target values from the training set.

$$X = \begin{pmatrix} — & (x^{(1)})^T & — \\ — & (x^{(2)})^T & — \\ — & \vdots & — \\ — & (x^{(n)})^T & — \end{pmatrix}, \quad y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{pmatrix}$$

Additionally, if we included the intercept terms $x_0^{(i)} = 1$, then we would have an $n \times (d+1)$ matrix. Now, finding the linear function h that minimizes J is to solve the least squares solution to the linear equation

$$X\theta = y$$

With a bit of linear algebra, this is equivalent to solving the normal equation for θ:

$$X^T X\theta = X^T y \implies \theta = (X^T X^{-1})X^T y$$

7.1.3 Probabilistic Interpretation

In this section we give a set of probabilistic assumptions under which least-squares regression (and the cost function J) is derived as a very natural algorithm. First, assume that the target variables and the inputs are related via the equation

$$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)}$$

where $\epsilon^{(i)}$ is an error term that captures either unmodeled effects or random noise. Let us further assume that the $\epsilon^{(i)}$ are distributed IID according to a Gaussian distribution with mean 0 and some variance σ^2. That is,

$$\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$$

and the density p is given by

$$p\big(\epsilon^{(i)}\big) = \frac{1}{\sigma\sqrt{2\pi}} \exp\bigg(- \frac{(\epsilon^{(i)})^2}{2\sigma^2}\bigg)$$

These are reasonably justifiable assumptions, since colloquially, we are saying that the effects of the errors should be the same for each training sample. This means that the distribution of $y^{(i)}$ given $x^{(i)}$ and parameterized by θ is just a simple shift of the distribution by $\theta^T x^{(i)}$:

$$y^{(i)} \,|\, x^{(i)}; \theta \sim \mathcal{N}(\theta^T x^{(i)}, \sigma^2) \implies p\big(y^{(i)} \,|\, x^{(i)}; \theta\big) = \frac{1}{\sigma\sqrt{2\pi}} \exp\bigg(- \frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\bigg)$$

Since for all $i = 1, \dots, n$, $y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)}$ is a Gaussian distribution, we can create a joint distribution of all the $y^{(i)} \,|\, x^{(i)}; \theta$ to create a **multivariate Gaussian distribution**:

$$y = X\theta + \epsilon, \quad \epsilon \in \mathcal{P}^n$$

where $\mathcal{P}$ is the space of probability distributions. By independence, this multivariate Gaussian distribution has density that is created by multiplying the individual densities of each element $\epsilon^{(i)}$, which is viewed as a function of y. However, if we would like to view it as a function of θ, then we fix the X, y at the observed values and call this the **likelihood function** $L(\theta)$:

$$\begin{aligned} L(\theta) = p(y \mid X; \theta) &= \prod_{i=1}^{n} p\big(y^{(i)} \mid x^{(i)}; \theta\big) \\ &= \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(- \frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2} \right) \end{aligned}$$

That is, L takes in a value of θ that represents a certain linear best fit model h, and it tells us the probability of output y happening given inputs X in the form of a density value.

Now, given this probabilistic model relating the $y^{(i)}$s to the $x^{(i)}$s, the principle of *maximum likelihood* says that we should choose θ such that $L(\theta)$ is as high as possible so that we would get the θ value that has the greatest probability of outputting y given data X. But this is the same as maximizing the **log likelihood** $l(\theta)$:

$$\begin{aligned} l(\theta) &= \log L(\theta) \\ &= \log \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(- \frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2} \right) \\ &= \sum_{i=1}^{n} \log \frac{1}{\sigma\sqrt{2\pi}} \exp\left(- \frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2} \right) \\ &= n \log \frac{1}{\sigma\sqrt{2\pi}} - \frac{1}{\sigma^2} \cdot \frac{1}{2} \sum_{i=1}^{n} \big(y^{(i)} - \theta^T x^{(i)}\big)^2 \end{aligned}$$

Hence, maximizing $l(\theta)$ gives the same answer as maximizing

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{n} \big(y^{(i)} - \theta^T x^{(i)}\big)^2$$

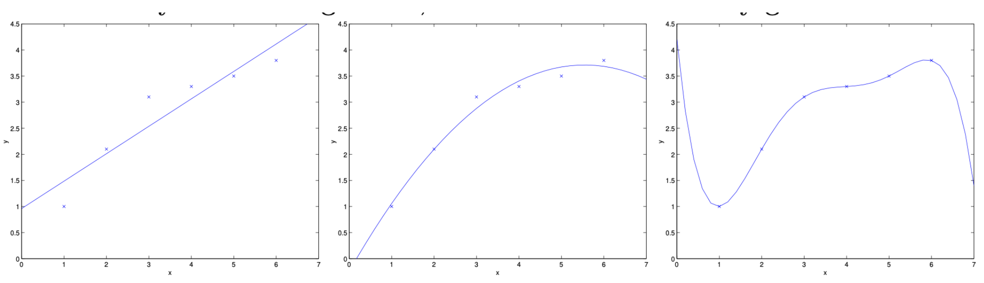
Therefore, under the previous probabilistic assumptions on the data, least-squares regression corresponds to finding the maximum likelihood estimate of θ. Note that our final choice of θ does not depend on σ.

7.1.4 Locally Weighted Linear Regression

However, not all data sets may fit with a line. For example, we can fit a quadratic model of form

$$y = \theta_0 + \theta_1 x + \theta_2 x^2$$

A higher-dimensional model can be made if our current model is **underfitting** (in which the data clearly shows structure not captured by the model). However, **overfitting** is not good either.



Another type of regression algorithm is called the **locally weighted linear regression (LWR)** which puts a weight on each feature and then attempts to minimize the cost function below:

$$\sum_i \left(y^{(i)} - \theta^T x^{(i)}\right)^2 \implies \sum_i w^{(i)} \left(y^{(i)} - \theta^T x^{(i)}\right)^2$$

where the $w^{(i)}$s are non-negative valued **weights** which determine how "important" a certain parameter is. A fairly standard choice for the weights is

$$w^{(i)} = \exp\left(- \frac{(x^{(i)} - x)^T (x^{(i)} - x)}{2\tau^2} \right) \text{ or } w^{(i)} = \exp\left(- \frac{(x^{(i)} - x)^T \Sigma^{-1} (x^{(i)} - x)}{2\tau^2} \right)$$

for adjusted values of τ and Σ, where τ is called the **bandwidth** parameter. Note that the weights depend on the particular point x at which we're trying to evaluate x. That is, we must determine beforehand what x is. Furthermore,

1. If $||x^{(i)} - x||$ is small, then $w^{(i)}$ is close to 1, and if $||x^{(i)} - x||$ is large, then $w^{(i)}$ is small. Hence, θ is chosen giving a much higher "weight" to the training example close to the query point x.
2. τ controls how quickly the weight of a training example falls off with distance of its $x^{(i)}$ from the query point x.

Parametric vs Nonparametric Algorithms

The unweighted linear regression algorithm is an example of an **parametric learning algorithm** because it has a fixed, finite number of parameters θ_i which are fit to the data. Once we've fit the θ_s's and stored them away, we no longer need to keep the training data around to make future predictions.

In contrast, to make predictions using locally weighted linear regression (a **non-parametric algorithm**), we need to keep the entire training set around.

7.2 Classificiation and Logistic Regression

Regression algorithms attempt to predict target variables that are continuous. But if y can take on only a small number of discrete values, thi sis a **classification problem**. We focus on the **binary classification** problem in which y can only take on two values 0 or 1. Given $x^{(i)}$, the corresponding $y^{(i)}$ is also called the **label** for the training example.

7.2.1 Logistic Regression

We could approach the classification problem ignoring the fact that y is discrete-valued and use our old linear regression algorithm to try to predict y given x. But since $h_\theta(x)$

takes values in $\{0,1\}$, we can adjust our hypothesis h to the logistic equation

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}, \text{ where } g(z) = \frac{1}{1+e^{-z}}$$

which has the range $(0,1)$, and $g(z) \to 1$ as $z \to \infty$ and and $g(z) \to 0$ as $z \to -\infty$. Note that given the logistic regression model, we can fit the appropriate θ for it using the maximum likelihood under a set of assumptions. Assume that we have a nonlinear equation $h : \mathbb{R}^n \longrightarrow [0,1]$ such that:

$$\begin{aligned} \mathbb{P}(y = 1 \mid x; \theta) &= h_\theta(x) \\ \mathbb{P}(y = 0 \mid x; \theta) &= 1 - h_\theta(x) \end{aligned}$$

which can be written more compactly as

$$p(y \mid x; \theta) = \big(h_\theta(x)\big)^y \big(1 - h_\theta(x)\big)^{1-y}$$

Assuming that the n training examples were generated independently, we can then write down the likelihood of the parameters as

$$\begin{aligned} L(\theta) &= p(y \mid X; \theta) \\ &= \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}; \theta) \\ &= \prod_{i=1}^{n} \big(h_\theta(x^{(i)})\big)^{y^{(i)}} \big(1 - h_\theta(x^{(i)})\big)^{1-y^{(i)}} \end{aligned}$$

As before, it is simpler to maximize the log likelihood

$$\begin{aligned} l(\theta) &= \log L(\theta) \\ &= \sum_{i=1}^{n} y^{(i)} \log h(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \end{aligned}$$

using gradient *ascent* and its rule

$$\theta = \theta + \nabla_\theta l(\theta)$$

But since $g'(z) = g(z)\,(1 - g(z))$, we can derive

$$\begin{aligned} \frac{\partial}{\partial \theta_j} l(\theta) &= \left(y \frac{1}{g(\theta^T x)} - (1-y) \frac{1}{1 - g(\theta^T x)} \right) \frac{\partial}{\partial \theta_j} g(\theta^T x) \\ &= \left(y \frac{1}{g(\theta^T x)} - (1-y) \frac{1}{1 - g(\theta^T x)} \right) g(\theta^T x)(1 - g(\theta^T x)) \frac{\partial}{\partial \theta_j} \theta^T x \\ &= \big(y(1 - g(\theta^T x)) - (1-y) g(\theta^T x) \big) x_j \\ &= \big(y - h_\theta(x) \big) x_j \end{aligned}$$

Therefore, the stochastic gradient ascent rule is reduced to (vector and coordinate form)

$$\theta = \theta + \alpha\big(y^{(i)} - h_\theta(x^{(i)}\big)\, x^{(i)} \iff \theta_j = \theta_j + \alpha\big(y^{(i)} - h_\theta(x^{(i)})\big)\, x_j^{(i)}$$

7.2.2 The Perceptron Learning Algorithm

We can modify the logistic regression method to "force" it to output values that are either 0 or 1 exactly. To do so, we can change the definition of g to be the threshold function

$$g(z) \equiv \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

If we then let $h_\theta(x) = g(\theta^T x)$, then h would be a function that outputs 1 whenever the value $\theta^T x$ reaches a certain threshold. If we use the update rule

$$\theta_j = \theta_j + \alpha\big(y^{(i)} - h_\theta(x^{(i)})\big)\, x_j^{(i)}$$

then we have the **perceptron learning algorithm**.

7.2.3 Newton-Raphson Method

A simple algorithm for finding the zero of a function $f : \mathbb{R} \longrightarrow \mathbb{R}$ is to perform the following update:

$$\theta = \theta - \frac{f(\theta)}{f'(\theta)}$$

If we wanted to maximize some function l, we would just need to find the critical points of l, which could be done by

$$\theta = \theta - \frac{l'(\theta)}{l''(\theta)}$$

Generalizing this to vector-valued functions $l_\theta : \mathbb{R}^d \longrightarrow \mathbb{R}$, the multidimensional version of Newton's method (called the Newton-Raphson method) is given by

$$\theta = \theta - H^{-1}\nabla_\theta l(\theta)$$

where H is the Hessian matrix of l_θ and $\nabla_\theta l(\theta)$ is vector of partial derivatives of l. Newton's method typically enjoys faster convergence than batch gradient descent$>$ However, it can be more computationally expensive since it requires finding and inverting the $d \times d$ Hessian, but as long as d is not too large, it is usually much faster overall.

7.3 Generalized Linear Models (GLMs)

A class of distributions is in the **exponential family** if it can be written in the form

$$p(y; \eta) = b(y) \exp\big(\eta^T T(y) - a(\eta)\big)$$

That is, given a *fixed* natural parameter η, $p(y; \eta)$ (y parameter) gives a distribution that outputs the probability of getting y. A few things to explain here:

1. η is called the **natural** or **canonical parameter**
2. $T(y)$ is called the **sufficient statistic**. In most cases, we will consider $T(y) = y$.
3. $a(\eta)$ is the **log partition function**, where the quantity $e^{-a(\eta)}$ is a normalization constant such that the distribution sums/integrates to 1.

A fixed choice of T, a, b defines a **family** of distributions that is parameterized by η.

Lemma 7.3.1. The Bernoulli and Gaussian distributions are both in the exponential family.

Proof. Assume $y|x;\theta \sim \text{Bernoulli}(\phi)$. Then,

$$\begin{aligned} p(y;\phi) &= \phi^y(1-\phi)^{1-y} \\ &= \exp\big(y\log\phi + (1-y)\log(1-\phi)\big) \\ &= \exp\left(\left(\log\Big(\frac{\phi}{1-\phi}\Big)\right)y + \log\big(1-\phi\big)\right) \end{aligned}$$

Thus, the natural parameter is $\eta = \log\big(\phi/(1-\phi)\big)$, $T(y) = y$, $a(\eta) = -\log(1-\phi) = \log(1+e^\eta)$, and $b(y) = 1$. As for the Gaussian, its full form is actually multivariate if we treat σ^2 as a variable, but treating it as $\sigma = 1$, we get

$$\begin{aligned} p(y;\mu) &= \frac{1}{\sqrt{2\pi}}\exp\Big(-\frac{1}{2}(y-\mu)^2\Big) \\ &= \frac{1}{\sqrt{2\pi}}\exp\Big(-\frac{1}{2}y^2\Big)\cdot\exp\Big(\mu y - \frac{1}{2}\mu^2\Big) \end{aligned}$$

Thus, we have $\eta = \mu$, $T(y) = y$, $a(\eta) = \mu^2/2$, and $b(y) = \frac{1}{\sqrt{2\pi}}\exp(-y^2/2)$. ■

Some other distributions in the exponential family are:

1. Multinomial
2. Poisson
3. Gamma, Exponential
4. Beta, Dirichlet

7.3.1 Constructing GLMs

When considering a regression or classification problem where we would like to predict the value of some random variable y as a function of x. To derive a GLM for this problem, we will make the following assumptions about the conditional distribution of y given x and about our model:

1. $y \mid x;\theta \sim \text{ExponentialFamily}(\eta)$. That is, given x and θ, the distribution of y follows some exponential family distribution with parameter η.
2. Given x, our goal is to predict the expected value of $T(y)$ given x. But since in most cases, $T(y) = y$, this means that we would like our hypothesis h to satisfy

$$h(x) = \mathbb{E}\big(y \mid x\big)$$

3. The natural parameter η and the inputs x are related linearly. We can just simply think of

$$\eta = \theta^T x$$

   This is the least intuitive of our assumptions, so we can think of this as a design choice.

Ordinary Least Squares

To show that ordinary least squares is a special case of the GLM family of models, note that the output y, given input x, has a Gaussian distribution. Let $\mu = \theta^T x^{(i)}$ and let

$$y^{(i)} \mid x^{(i)}; \theta \sim \mathcal{N}(\mu, \sigma^2)$$

Since this Gaussian distribution is in the exponential family, it follows that $\mu = \eta$ and we have

$$\begin{aligned} h_\theta(x) &= \mathbb{E}(y \,|\, x; \theta) \\ &= \mu \\ &= \eta \\ &= \theta^T x \end{aligned}$$

To summarize, we first assume the condition that the output y, given an input x, is randomly distributed Gaussian. This means that for every single vector x, we have a Gaussian distribution $y \,|\, x; \theta$ corresponding to that x. We can simply take the expectation of all of these normal distributions to create a function $h_\theta(x)$ which takes in x, looks at the conditional distribution $y \,|\, x; \theta$, and outputs its expected value. But the expected value of a Normal distribution is just its mean μ, so we have

$$h_\theta(x) = \mathbb{E}(y \,|\, x; \theta) = \mu$$

These final steps are the least intuitive. We find out that the natural parameter η of this exponential distribution, is, in this case, the μ parameter. By assumption three we just let $eta = \theta^T x$, and we are left with

$$h_\theta(x) = \theta^T x$$

and we have found that the form of the hypothesis function is $\theta^T x$. *Now* that we have have found this form, we can simply use gradient descent to minimize the cost function and get the best fit.

Logistic Regression

Being interested in binary classification, we let $y \in \{0, 1\}$ and choose the Bernoulli family of distributions to model the conditional distribution of y given x. In our formulation of the Bernoulli distribution as an exponential family distribution, we had $\phi = 1/(1 + e^{-\eta})$. Furthermore, note that if $y \,|\, x; \theta \sim \text{Bernoulli}(\phi)$, then $\mathbb{E}(y \,|\, x; \theta) = \phi$. So, we get

$$\begin{aligned} h_\theta(x) &= \mathbb{E}(y \,|\, x; \theta) \\ &= \phi \\ &= \frac{1}{1 + e^{-\eta}} \\ &= \frac{1}{1 + e^{-\theta^T x}} \end{aligned}$$

This gives us hypothesis functions of the form $h_\theta(x) = 1/(1 + e^{-\theta^T x})$. Therefore, we can say that once we assume that y conditioned on x is Bernoulli, it arises as a consequence of the definition of GLMs and exponential family distributions that the hypothesis is logistic.

The function g giving the distribution's mean as a function of the natural parameter

$$g(\eta) = \mathbb{E}\big(T(y); \eta\big)$$

is called the **canonical response function**. It's inverse, g^{-1}, is called the **canonical link function**. Thus, the canonical response fuction for the Gaussian family is the identity function, and the canonical response function for the Bernoulli is the logistic function.

Softmax Regression

Consider a classification problem in which the response variable y can take on any one of k values. That is,

$$\mathcal{Y} = \{1, 2, ..., k\}$$

The response variable is still discrete, and we can model it as distributed according to a multinomial distribution, which can be thought of as the sum of k iid distributions M where

$$\mathbb{P}(M = i) = \phi_i, \quad \sum_{i=1}^{k} \phi_i = 1$$

We can parameterize the multinomial distribution over k possible outcomes with $k-1$ parameters $\phi_1, \phi_2, ..., \phi_{k-1}$ specifying the probability of each of the outcomes and one final determined ϕ_k (since all ϕ_is must sum to 1). Therefore,

$$\phi_i = p(y = i\ \phi)\ \ (i = 1, ..., k-1), \quad \phi_k = p(y = k; \phi) = 1 - \sum_{i=1}^{k-1} \phi_i$$

To express the multinomial as an exponential family distribution, we define $T(y) \in \mathbb{R}^{k-1}$ as

$$T(1) = e_1,\ T(2) = e_2,\ ..., T(k-1) = e_{k-1},\ T(k) = 0$$

where e_i denotes the canonical ith basis vector and 0 is the zero vector. Note that $T(y) \neq y$; in fact, it isn't even a real number, it is a $k-1$ dimensional vector. We denote $(T(y))_i$ to denote the ith element of the vector $T(y)$.

We also introduce the **indicator function** $1\{\cdot\}$ which takes on the value of 1 if the argument is true, and 0 otherwise. That is,

$$1\{True\} = 1,\ \ 1\{False\} = 0$$

For example, $1\{2 = 3\} = 0, 1\{3 = 5-2\} = 1$. So, we can write the relationship between $T(y)$ and y as

$$\big(T(y)\big)_i = 1\{y = i\}$$

which basically means that the probability that the ith element in the vector $(T(y))_i$ is 1 and 0 for every other element in the vector. Furthermore, we have that $\mathbb{E}\big((T(y))_i\big) =$

$P(y = i) = \phi_i$ since this is Bernoulli. With this, we can derive

$$\begin{aligned}
p(y; \phi) &= \phi_1^{1\{y=1\}} \phi_2^{1\{y=2\}} \dots \phi_k^{1\{y=k\}} \\
&= \phi_1^{1\{y=1\}} \phi_2^{1\{y=2\}} \dots \phi_k^{1-\sum_{i=1}^{k-1} 1\{y=i\}} \\
&= \phi_1^{(T(y))_1} \phi_2^{(T(y))_2} \dots \phi_k^{1-\sum_{i=1}^{k-1} (T(y))_i} \\
&= \exp\left((T(y))_1 \log(\phi_1) + (T(y))_2 \log(\phi_2) + \dots \left(1 - \sum_{i=1}^{k-1} (T(y))_i\right) \log(\phi_k)\right) \\
&= \exp\left((T(y))_1 \log(\phi_1/\phi_k) + \dots + (T(y))_{k-1} \log(\phi_{k-1}/\phi_k) + \log(\phi_k)\right) \\
&= b(y) \exp\left(\eta^T T(y) - a(\eta)\right)
\end{aligned}$$

where

$$\eta = \begin{pmatrix} \log(\phi_1/\phi_k) \\ \log(\phi_2/\phi_k) \\ \vdots \\ \log(\phi_{k-1}/\log_k) \end{pmatrix}, \ a(\eta) = -\log(\phi_k), \ b(y) = 1$$

hence making it an exponential distribution. The link function is given by

$$\eta_i = \log \frac{\phi_i}{\phi_k}$$

To invert the link function to derive the response function we have

$$\begin{aligned}
e^{\eta)i} = \frac{\phi_i}{\phi_k} &\implies \phi_k e^{\eta_i} = \phi_i \implies \phi_k \sum_{i=1}^k e^{\eta_i} = \sum_{i=1}^k \phi_i = 1 \\
&\implies \phi_k = \frac{1}{\sum_{i=1}^k e^{\eta_i}} \\
&\implies \phi_i = \frac{e^{\eta_i}}{\sum_{j=1}^k e^{\eta_j}}
\end{aligned}$$

The function mapping the ηs to the ϕs is called the **softmax** function. Using assumption 3 again, our model assumes that the conditional distribution of y given x is given by:

$$\begin{aligned}
p(y = i \mid x; \theta) &= \phi_i \\
&= \frac{e^{\eta_i}}{\sum_{j=1}^k e^{\eta_j}} \\
&= \frac{e^{\theta_i^T x}}{\sum_{j=1}^k e^{\theta_j^T x}}
\end{aligned}$$

This model, which applies to classification problems where $y \in \{1, 2, \dots, k\}$ is called **softmax regression**. It is a generalization of logistic regression. Our hypothesis will output

$$
\begin{aligned}
h_\theta(x) &= \mathbb{E}\big(T(y) \,|\, x; \theta\big) \\
&= \mathbb{E}\left(\begin{array}{c|c} 1\{y=1\} & \\ 1\{y=2\} & \\ \vdots & x;\theta \\ 1\{y=k-2\} & \\ 1\{y=k-1\} & \end{array} \right) \\
&= \begin{pmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{k-1} \end{pmatrix} = \begin{pmatrix} \frac{\exp(\theta_1^T x)}{\sum_{j=1}^k \exp(\theta_j^T x)} \\ \frac{\exp(\theta_1^T x)}{\sum_{j=1}^k \exp(\theta_j^T x)} \\ \vdots \\ \frac{\exp(\theta_1^T x)}{\sum_{j=1}^k \exp(\theta_j^T x)} \end{pmatrix}
\end{aligned}
$$

That is, our hypothesis will output the estimated probability that $p(y = i \,|\, x; \theta)$ for every value of $i = 1, \ldots k$. Even though $h_\theta(x)$ as defined above is only $k-1$ dimensional, clearly $p(y = k \,|\, x; \theta)$ can be obtained as $1 - \sum_{i=1}^{k-1} \phi_i$.

As for parameter fitting, if we would like to learn the parameters θ_i of this model, we would write down the log likelihood

$$
\begin{aligned}
l(\theta) &= \sum_{i=1}^n \log p(y^{(i)} \,|\, x^{(i)}; \theta) \\
&= \sum_{i=1}^n \log \prod_{l=1}^k \left(\frac{e^{\theta_l^T x(i)}}{\sum_{j=1}^k e^{\theta_j^T x^{(i)}}} \right)^{1\{y^{(i)}=l\}}
\end{aligned}
$$

and maximizing $l(\theta)$ with respect to θ using a method such as gradient ascent or Newton's method.

7.4 Generative Learning Algorithms

We have dealt with learning algorithms that model $p(y \,|\, x; \theta)$: the conditional distribution of y given x. For instance, logistic regression modeled $p(y \,|\, x; \theta)$ as $h_\theta(x) = g(\theta^T x)$ where g is the sigmoid function. Here is a different type of learning algorithm.

Consider a classification problem in which we want to learn to distinguish between elephants ($y = 1$) and dogs ($y = 0$), based on some features of an animal. Given a training set, an algorithm like logistic regression or the perceptron algorithm (basically) tries to find a straight line—that is, a decision boundary—that separates the elephants and dogs. Then, to classify a new animal as either an elephant or a dog, it checks on which side of the decision boundary it falls, and makes its prediction accordingly.

Here's a different approach. First, looking at elephants, we can build a model of what elephants look like. Then, looking at dogs, we can build a separate model of what dogs look like. Finally, to classify a new animal, we can match the new animal against the elephant model, and match it against the dog model, to see whether the new animal looks more like the elephants or more like the dogs we had seen in the training set.

Algorithms that try to learn p(y|x) directly (such as logistic regression), or algorithms that try to learn mappings directly from the space of inputs X to the labels 0, 1, (such as the perceptron algorithm) are called discriminative learning algorithms. Here, we'll talk about algorithms that instead try to model p(x|y) (and p(y)). These algorithms are called generative learning algorithms. For instance, if y indicates whether an example is a dog (0) or an elephant (1), then $p(x \mid y = 0)$ models the distribution of dogs' features, and $p(x \mid y = 1)$ models the distribution of elephants' features.

7.4.1 Gaussian Discriminant Analysis Model

For classification problems where the input features x are continuous-valued random variables, we can use the Gaussian Discriminant Analysis (GDA) model, which models $p(x \mid y)$ using a multivariate normal distribution. The model is

$$\begin{aligned} y &\sim \text{Bernoulli}(\phi) \\ x \mid y = 0 &\sim \mathcal{N}(\mu_0, \Sigma) \\ x \mid y = 1 &\sim \mathcal{N}(\mu_1, \Sigma) \end{aligned}$$

Writing out the distributions, this is:

$$\begin{aligned} p(y) &= \phi^y (1-\phi)^{1-y} \\ p(x \mid y = 0) &= \frac{1}{(2\pi)^{d/2} |\Sigma|^{1/2}} \exp\Big(-\frac{1}{2}(x-\mu_0)^T \Sigma^{-1} (x-\mu_0) \Big) \\ p(x \mid y = 1) &= \frac{1}{(2\pi)^{d/2} |\Sigma|^{1/2}} \exp\Big(-\frac{1}{2}(x-\mu_1)^T \Sigma^{-1} (x-\mu_1) \Big) \end{aligned}$$

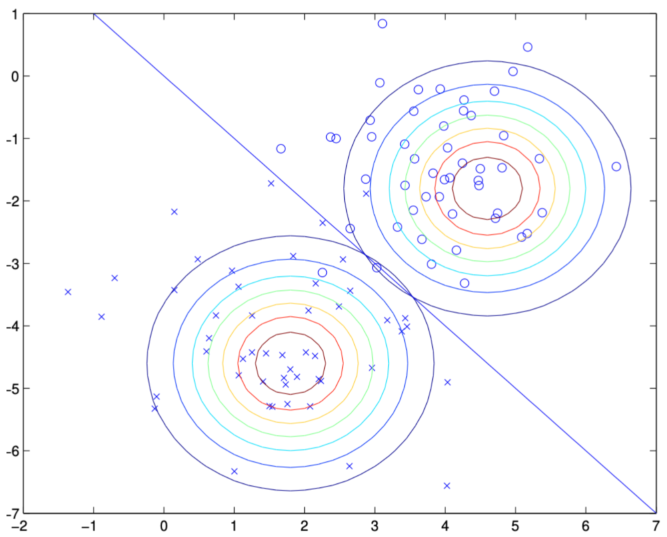
Note that while there are two different mean vectors μ_0 and μ_1, this model is usually applied using only one covariance matrix Σ. The parameters of our model are $\phi, \Sigma, \mu_0, \mu_1$. The log-likelihood of the data is give by

$$\begin{aligned} l(\phi, \mu_0, \mu_1, \Sigma) &= \log \prod_{i=1}^{n} p(x^{(i)}, y^{(i)}; \phi, \mu_0, \mu_1, \Sigma) \\ &= \log \prod_{i=1}^{n} p(x^{(i)} \mid y^{(i)}; \mu_0, \mu_1, \Sigma)\, p(y^{(i)}; \phi) \end{aligned}$$

By maximizing l with respect to the parameters, we find the maximum likelihood estimate of the parameters to be

$$\begin{aligned} \phi &= \frac{1}{n} \sum_{i=1}^{n} 1\{y^{(i)} = 1\} \\ \mu_0 &= \frac{\sum_{i=1}^{n} 1\{y^{(i)} = 0\} x^{(i)}}{\sum_{i=1}^{n} 1\{y^{(i)} = 0\}} \\ \mu_1 &= \frac{\sum_{i=1}^{n} 1\{y^{(i)} = 1\} x^{(i)}}{\sum_{i=1}^{n} 1\{y^{(i)} = 1\}} \\ \Sigma &= \frac{1}{n} \sum_{i=1}^{n} (x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{Y^{(i)}})^T \end{aligned}$$

Visually, we can imagine the algorithm going through each point that is a dog (the circles) or an elephant (the crosses) and constructing a 2-dimensional Gaussian by finding the maximum likelihood estimate of $\mu_0, \mu_1, \Sigma, \phi$. Note that ϕ is "maximized" by directly taking the portion of samples of y that have value 1. The end result looks something like this:



Note that we have constructed a straight line $\mathcal{L}$ representing the decision boundary at which $p(y = 1 \,|\, x \in \mathcal{L}) = p(y = 0 \,|\, x \in \mathcal{L}) = 0.5$. If a future sample has value on the left side of $\mathcal{L}$ it will be classified as a cross and if on the other side, it will be classified as a circle.

When comparing GDA to other models such as logistic regression, GDA makes strong modeling assumptions and is more data efficient (i.e. requires less training data to learn "well") when the modeling assumptions are at least approximately correct. Logistic regression makes weaker assumptions and is significantly more robust to deviations from modeling assumptions. Specifically, when the data is non-Gaussian, then in the limit of large datasets, logistic regressiom will almost always do better than GDA, which is why it's used more in practice.

7.4.2 Naive Bayes

Unlike GDA (where the feature vectors x were continuous valued), we work with ones in which the x_j's are discrete.

Say that we are building a email spam filter using machine learning that classifies emails to either spam or non-spam by reading the text. Let us represent an email with a vector whose length is equal to the number of words in the **dictionary**. This set of words encoded into this vector is called the **vocabulary**. We can literally go through the entire English dictionary and list the words, but in practice it is more common to look through our training set and encode in our feature vector only the words that occur at least once there. This saves space and allows us to include words unique to our email. If an email contains the jth word of the dictionary, we set $x_j = 1$ and other set $x_j = 0$.

Say that our dictionary contains 50000 elements. In order to model $p(x \,|\, y)$, then it is computationally infeasible to explicitly model $x \in \{0, 1\}^{50000}$ as a multinomial distribution. Therefore, we make a very strong assumption called the **Naive Bayes (NB)**

assumption that states that *the x_i's are conditionally independent given y.*

$$p(x_i \,|\, y) = p(x_i \,|\, y, x_j) \text{ for all } i, j$$

That is, if you knew that a particular email is spam ($y = 1$), then knowledge of x_i will have no effect on your beliefs about the value of x_j. Note that this is not the same as saying that x_i and x_j are independent. We have

$$\begin{aligned} p(x \,|\, y) &= p(x_1, \ldots, x_{50000} \,|\, y) \\ &= p(x_1 \,|\, y) p(x_2 \,|\, y, x_1) p(x_3 \,|\, y, x_1, x_2) \ldots p(x_{50000} \,|\, y, x_1, \ldots, x_{49999}) \\ &= p(x_1 \,|\, y) p(x_2 \,|\, y) p(x_3 \,|\, y) \ldots p(x_{50000} \,|\, y) \\ &= \prod_{j=1}^{50000} p(x_j \,|\, y) \end{aligned}$$

Note that even though the Naive Bayes assumption is extremely strong, the resulting algorithm works well on many problems. The model for our algorithm is:

$$\begin{aligned} y \sim \text{Bernoulli}(\phi) &\implies p(y) = \phi^y (1-\phi)^{1-y} \\ x_j \,|\, y = 0 \sim \text{Bernoulli}(\phi_{j|y=0}) &\implies p(y) = \phi_{j|y=0}^y (1-\phi_{j|y=0})^{1-y} \\ x_j \,|\, y = 1 \sim \text{Bernoulli}(\phi_{j|y=1}) &\implies p(y) = \phi_{j|y=1}^y (1-\phi_{j|y=1})^{1-y} \end{aligned}$$

Therefore, given the training set $\{(x^{(i)}, y^{(i)}); i = 1, \ldots n\}$, we can write down the joint likelihood of the data:

$$\mathcal{L}(\phi_y, \phi_{j|y=0}, \phi_{j|y=1}) = \prod_{i=1}^{n} p\big(x^{(i)}, y^{(i)}\big)$$

Maximizing this with respect to $\phi_y, \phi_{j|y=0}, \phi_{j|y=1}$ gives the maximum likelihood estimates:

$$\begin{aligned} \phi_{j|y=1} &= \frac{\sum_{i=1}^n 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^n 1\{y^{(i)} = 1\}} \\ \phi_{j|y=0} &= \frac{\sum_{i=1}^n 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^n 1\{y^{(i)} = 0\}} \\ \phi_y &= \frac{\sum_{i=1}^n 1\{y^{(i)} = 1\}}{n} \end{aligned}$$

The parameters have a very natural interpretation. For instances, $\phi_{j|y=1}$ is just the fraction of the spam ($y = 1$) emails in which the word j appears. To make a prediction on a new example with features x, we then simply calculate

$$\begin{aligned} p(y = 1|x) &= \frac{p(x \,|\, y- = 1) p(y = 1)}{p(x)} \\ &= \frac{\Big(\prod_{j=1}^d p(x_j \,|\, y = 1)\Big) p(y = 1)}{\Big(\prod_{j=1}^d p(x_j \,|\, y = 1)\Big) p(y = 1) + \Big(\prod_{j=1}^d p(x_j \,|\, y = 0)\Big) p(y = 0)} \end{aligned}$$

and pick whichever class has the higher posterior probability.

Note that we can extend this beyond binary valued features to those that can take values in $\{1, 2, \ldots, k_j\}$. For example, we can simply model $p(x_j \,|\, y)$ as multinomial rather than as Bernoulli. Additionally, if the original input value is continuous it is common to **discretize** it by turning it into a small set of discrete values and applying Naive Bayes.

Laplace Smoothing

There is a problem with the Naive Bayes algorithm that in some cases outputs 0/0 as a probability. For example, in our email spam filter, if we encounter a completely new word, say "neurips" as our 35000th word, our Naive Bayes spam filter would pick maximum likelihood estimates of the parameters to be

$$\begin{aligned}\phi_{35000|y=1} &= \frac{\sum_{i=1}^{n} 1\{x_{35000}^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^{n} 1\{y^{(i)} = 0\}} = 1 \\ \phi_{35000|y=0} &= \frac{\sum_{i=1}^{n} 1\{x_{35000}^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^{n} 1\{y^{(i)} = 0\}} = 0\end{aligned}$$

That is, because it has never seen neurips before in either spam or non-spam training examples, it thinks the probability of seeing it in either type of email is zero. Hence, the posterior probability turns out to be $p(y = 1 \mid x) = 0/0$, resulting in a problem.

Stating the problem more broadly, it is statistically a bad idea to esti- mate the probability of some event to be zero just because you haven't seen it before in your finite training set.

Given a multinomial random variable z taking values in $\{1, 2, \ldots, k\}$, we can parameterize our multinomial with $\phi_i = p(z = j)$. Given a set of n independent observations $\{z^{(1)}, \ldots, z^{(n)}\}$, the maximum likelihood estimates are given by

$$\phi_j = \frac{\sum_{i=1}^{n} 1\{z^{(i)} = j\}}{n}$$

But with this, some of the ϕ_j's might end up as 0. To avoid this, we can use **Laplace smoothing**, which replaces the above estimate with

$$\phi_j = \frac{1 + \sum_{i=1}^{n} 1\{z^{(i)} = j\}}{k + n}$$

This "smoothes" the probability such that every value of z has a nonzero probability of occurring.

By applying Laplace smoothing on our Naive Bayes classifier, we can obtain the following estimates of the parameters:

$$\begin{aligned}\phi_{j|y=1} &= \frac{1 + \sum_{i=1}^{n} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\}}{2 + \sum_{i=1}^{n} 1\{y^{(i)} = 1\}} \\ \phi_{j|y=0} &= \frac{1 + \sum_{i=1}^{n} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{2 + \sum_{i=1}^{n} 1\{y^{(i)} = 0\}}\end{aligned}$$

This modification takes care of the 0/0 problems.

Event Models for Text Classification

We will now describe the **Multinomial event model**. To describe this model, we use a different notation and set of features for representing emails. We let x_j denote the identity of the jth word in the email, taking values in $\{1, 2, \ldots |V|\}$, where V is the dictionary.

An email of d words is not represented by a vector $(x_1, x_2, \ldots, x_d)$ (note that d can vary for different documents).

In the multinomial event model, we assume that the way an email is generated is via a random process in which spam/non-spam is first determined (according to $p(y)$) as before. Then, the sender of the email writes the email by first generating x_1 from some multinomial distribution over words ($p(x_1 \,|\, y)$). Next, the second word x_2 is chosen independently of x_1 but from the same multinomial distribution, and similarly for x_3, x_4, and so on, until all d words of the email have been generated. Thus, the overall probability of a message is given by

$$p(y) \prod_{j=1}^{d} p(x_j \,|\, y)$$

Note that this is the same formula as before, but now $x_j \,|\, y$ is a mulitnomial, rather than a Bernoulli distribution. Our model is now

$$\begin{aligned} y &\sim \text{Bernoulli}(\phi_y) \implies p(y) = \phi_y \\ x|y = 0 &\sim \text{Multinomial}(\phi_{k|y=0}) \implies p(x_j = k \,|\, y = 0) = \phi_{k|y=0} \\ x|y = 1 &\sim \text{Multinomial}(\phi_{k|y=1}) \implies p(x_j = k \,|\, y = 1) = \phi_{k|y=1} \end{aligned}$$

with three parameters. Note that we have assumed that $p(x_j \,|\, y)$ is the same for all values of j (i.e. the distribution according to which a word is generated does not depend on its position j within the email).

If we are given a training set $\{x^{(i)}, y^{(i)}; i = 1, \ldots n\}$, where

$$x^{(i)} = \left(x_1^{(i)}, x_2^{(i)}, \ldots, x_{d_i}^{(i)}\right)$$

(d_i is the number of words in the ith training example), the likelihood of the data is given by

$$\begin{aligned} \mathcal{L}(\phi_y, \phi_{k|y=0}, \phi_{k|y=1}) &= \prod_{i=1}^{n} p(x^{(i)}, y^{(i)}) \\ &= \prod_{i=1}^{n} \left(\prod_{j=1}^{d_i} p(x_j^{(i)} \,|\, y; \phi_{k|y=0}, \phi_{k|y=1}) \right) p(y^{(i)}; \phi_y) \end{aligned}$$

Maximizing this yields the maximum likelihood estimates of the parameters:

$$\begin{aligned} \phi_{k|y=1} &= \frac{\sum_{i=1}^{n} \sum_{j=1}^{d_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 1\}}{\sum_{i=1}^{n} 1\{y^{(i)} = 1\} d_i} \\ \phi_{k|y=0} &= \frac{\sum_{i=1}^{n} \sum_{j=1}^{d_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 0\}}{\sum_{i=1}^{n} 1\{y^{(i)} = 0\} d_i} \\ \phi_y &= \frac{\sum_{i=1}^{n} 1\{y^{(i)} = 1\}}{n} \end{aligned}$$

If we apply Laplace smoothing, we add 1 to the numerators and $|V|$ to the denominators to obtain:

$$\begin{aligned} \phi_{k|y=1} &= \frac{1 + \sum_{i=1}^{n} \sum_{j=1}^{d_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 1\}}{|V| + \sum_{i=1}^{n} 1\{y^{(i)} = 1\} d_i} \\ \phi_{k|y=0} &= \frac{1 + \sum_{i=1}^{n} \sum_{j=1}^{d_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 0\}}{|V| + \sum_{i=1}^{n} 1\{y^{(i)} = 0\} d_i} \end{aligned}$$

While not the best classifying algorithm, the Naive Bayes classifier often works surprisingly well and is a good first thing to try, given its simplicity and ease of implementation.

7.5 Kernel Methods

We talked about generating a linear function that predicts the training data, but what happens if the output value y can be more accurately represented as a *non-linear* function of inputs x?

If we have a cubic function $y = \theta_3 x^3 + \theta_2 x^2 + \theta_1 x + \theta_0$ in x, we can interpret this as a linear function over the a different set of feature variables $\{x^3, x^2, x, 1\}$. That is, let $\phi : \mathbb{R} \longrightarrow \mathbb{R}^4$ be be defined

$$\phi(x) = \begin{pmatrix} 1 \\ x \\ x^2 \\ x^3 \end{pmatrix} \in \mathbb{R}^4$$

and let $\theta = \begin{pmatrix} \theta_0 & \theta_1 & \theta_2 & \theta_3 \end{pmatrix}^T$. Then, we can rewrite the cubic function in x as:

$$\theta_3 x^3 + \theta_2 x^2 + \theta_1 x + \theta_0 = \theta^T \phi(x)$$

While the terminology depends on the context and author, we will denote the original input value (the x) as the input **attributes** of a problem, and when the original input is mapped to some new set of quantities $\phi(x)$, we will call these new quantities the **feature** variables. The map ϕ is called the **feature map**.

7.5.1 LMS with Features

We will first derive the (batch) gradient descent algorithm for fitting the model $\theta^T \phi(x)$. Recall that the ordinary least square problem for fitting $\theta^T x$ is

$$\theta = \theta + \alpha \sum_{i=1}^{n} \big(y^{(i)} - \theta^T x^{(i)}\big) x^{(i)}$$

Let $\phi : \mathbb{R}^d \longrightarrow \mathbb{R}^p$ be a feature map that maps attribute $x \in \mathbb{R}^d$ to the features $\phi(x) \in \mathbb{R}^p$ (in our previous example, $d = 1$ and $p = 4$). Now, our goal is to fit the function $\theta^T \phi(x)$, with θ being a vector in $\mathbb{R}^p$ instead of $\mathbb{R}^d$. We replace all occurrences of $x^{(i)}$ in the algorithm above by $\phi(x^{(i)})$ to get

$$\theta = \theta + \alpha \sum_{i=1}^{n} \big(y^{(i)} - \theta^T \phi(x^{(i)})\big) x^{(i)}$$

Similarly, the stochastic gradient descent update rule is:

$$\theta = \theta + \alpha \big(y^{(i)} - \theta^T \phi(x^{(i)}\big)\, \phi(x^{(i)}$$

7.5.2 LMS with the Kernel Trick

The gradient descent update above becomes computationally expensive when the features $\phi(x)$ is high-dimensional. For example, let $x \in \mathbb{R}^d$ and $\phi(x)$ be the vector that contains all the monomials of x with degree ≤ 3:

$$\phi(x) = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_1^2 \\ x_1x_2 \\ \vdots \\ x_1^3 \\ \vdots \end{pmatrix}$$

The dimension of the features $\phi(x)$ is of order $O(d^3)$, so given a d-dimensional input attribute x, we will have to compute and store at least d^k values in order to create the d^k-dimensional feature vector containing monomials of x with degree $\leq k$. It may seem that this d^3 runtime per update and memory usage are inevitable because the vector θ itself is of dimension $p \approx d^k$ and we may need to update every entry of θ and store it. However, the **kernel trick** allows us to significantly improve the runtime by not needing to store θ explicitly.

For simplicity, we assume (but need not) the initial value of $\theta = 0$ and run the batch GD update:

$$\theta = \theta + \alpha \sum_{i=1}^{n} \big(y^{(i)} - \theta^T x^{(i)}\big) x^{(i)}$$

Note that since θ is a vector, it can be represented as a linear combination of the vectors $\phi(x^{(1)}), \ldots, \phi(x^{(n)})$. It is easy to see this since every $(y^{(i)} - \theta^T \phi(x^{(i)}))$ is a constant linear coefficient of $\phi(x^{(i)}$ in the update rule. At initialization, $\theta = 0 = \sum_{i=1}^{n} 0 \cdot \phi(x^{(i)})$. Now, assuming that θ acn be represented as

$$\theta = \sum_{i=1}^{n} \beta_i \phi(x^{(i)})$$

for some $\beta_1, \ldots, \beta_n \in \mathbb{R}$, then we claim that θ is still a linear combination of the $\phi(x^{(i)})$'s.

$$\begin{aligned}
\theta &= \theta + \alpha \sum_{i=1}^{n} \big(y^{(i)} - \theta^T \phi(x^{(i)})\big) \phi(x^{(i)} \\
&= \sum_{i=1}^{n} \beta_i \phi(x^{(i)}) + \alpha \sum_{i=1}^{n} \big(y^{(i)} - \theta^T \phi(x^{(i)})\big) \phi(x^{(i)} \\
&= \sum_{i=1}^{n} (\beta_i + \alpha\big(y^{(i)} - \theta^T \phi(x^{(i)})\big)) \phi(x^{(i)}
\end{aligned}$$

Therefore, we can implicitly represent the p-dimensional vector θ by a set of coefficients $\beta_1, \ldots, \beta_n$. Towards doing this, we derive the update rule of the coefficients β_i's.

$$\beta_i = \beta_i + \alpha\big(y^{(i)} - \theta^T \phi(x^{(i)})\big)$$

Since we are updating θ with β_i's, we replace the θ with its old value $\theta = \sum_{j=1}^n \beta_j \phi(x^{(j)})$

$$\beta_i = \beta_i + \alpha\left(y^{(i)} - \sum_{j=1}^n \beta_j \phi(x^{(j)})^T \phi(x^{(i)})\right)$$

We see that $\phi(x^{(j)})^T\phi(x^{(i)}) = \langle \phi(x^{(j)}), \phi(x^{(i)})\rangle$ where $\langle \cdot, \cdot \rangle$ represents the Euclidean inner product of the two vectors.

Therefore, we have reduced the batch gradient descent algorithm into an algorithm that updates the value of β iteratively. We also need to compute the values of $\langle \phi(x^{(j)}), \phi(x^{(i)})\rangle$ for all pairs of i, j (which may take roughly $O(p)$ operations). However, note that

1. We can pre-compute the pairwise inner products $\langle \phi(x^{(j)}, \phi(x^{(i)}\rangle$ before the loop starts.
2. For the feature map ϕ, computing the inner product can be efficient since:

$$\begin{aligned}
\langle \phi(x), \phi(z)\rangle &= 1 + \sum_{i=1}^d x_i z_i + \sum_{i,j \in \{1,\ldots,d\}} x_i x_j z_i z_j + \sum_{i,j,k \in \{1,\ldots,d\}} x_i x_j x_k z_i z_j z_k \\
&= 1 + \sum_{i=1}^d x_i z_i + \left(\sum_{i=1}^d x_i z_i\right)^2 + \left(\sum_{i=1}^d x_i z_i\right)^3 \\
&= 1 + \langle x, z\rangle + \langle x, z\rangle^2 + \langle x, z\rangle^3
\end{aligned}$$

   Therefore, to compute $\langle \phi(x), \phi(z)\rangle$ we can first compute $\langle x, z\rangle$ with $O(d)$ time and then take another constant number of operations to compute $1 + \langle x, z\rangle + \langle x, z\rangle^2 + \langle x, z\rangle^3$.

Now, we define the **Kernel** corresponding to the feature map ϕ as a function that maps $\mathcal{X} \times \mathcal{X} \longrightarrow \mathbb{R}$ (where $\mathcal{X}$ is the input space, in our case $\mathcal{X} = \mathbb{R}^4$) satisfying

$$K(x, z) = \langle \phi(x), \phi(z)\rangle$$

Summary of Kernel Algorithm

We first compute all the values $K(x^{(i)}, x^{(j)}) \equiv \langle \phi(x^{(i)}), \phi(x^{(j)})\rangle$ for all $i, j \in \{1, \ldots, n\}$. Let K be the $n \times n$ matrix with $K_{ij} = K(x^{(i)}, x^{(j)})$ and set $\beta = 0$. We loop through this update step for all i's

$$\beta_i = \beta_i + \alpha\left(y^{(i)} - \sum_{j=1}^n \beta_j K(x^{(i)}, x^{(j)})\right) \iff \beta = \beta + \alpha(y + K\beta)$$

We can update the representation β of the vector θ efficiently with $O(n^2)$ time per update, a huge improvement. Now, once convergence requirements are met, we can get $\theta^T\phi(x)$ back with the calculations

$$\theta^T\phi(x) = \sum_{i=1}^n \beta_i \phi(x^{(i)})^T \phi(x) = \sum_{i=1}^n \beta_i K(x^{(i)}, x)$$

Therefore, all we really need to know about the feature map $\phi(x)$ is encapsulated in the corresponding kernel function $K(\cdot, \cdot)$.

Chapter 8

Quantum Computing

Chapter 9

HTML, CSS, JavaScript

HTML, which stands for the *Hyper Text Markup Language*, is the standard markup language for creating Web pages. It describes the structure of a webpage. The purpose of a **web browser** (Chrome, Safari, ...) is to read HTML documents and display them correctly.

Definition 9.0.1. AN **HTML element** is defined by a start tag, some content, and an end tag.

Even though professional IDEs can be used to create and modify webpages, simple text editors such as TextEdit are also available. You just have to type in the html code into a document and save it as a `.htm` or `.html` file.

9.1 HTML Tags

1. All HTML documents must start with a document type declaration: `<!DOCTYPE html>`. The `<!DOCTYPE>` declaration represents the document type and helps browsers to display webpages correctly. The `<DOCTYPE>` declaration for HTML5 (latest version of HTML) is:

   `<!DOCTYPE html>`

2. The HTML document itself begins with `<html>` and ends with `</html>`. The `lang` attribute for this tag is used to declare the language (the first two letters) of the Web page and to assist search engines and browsers. The country code (last two letters) can also be added to define the country.

```
<!DOCTYPE html>
<html lang="en">    #or <html lang="en-US">
<body>
...
</body>
</html>
```

3. The visible part of the HTML document is between `<body>` and `</body>`.

Definition 9.1.1. Some common HTML tags are:

1. **HTML headings**: `<h1> </h1>` (the most important heading), `<h2> </h2>`, `<h3> </h3>`, `<h4> </h4>`, `<h5> </h5>`, `<h6> </h6>` (The least important heading).

   (a) Each HTML heading has a default size, but you can specify the size for any heading with the `style` attribute, using the CSS `font-size` property:

   ```
   <h1 style="font-size:60px;">Heading 1</h1>
   ```

2. **HTML paragraphs**:

   ```
   <p> This is the first sentence in the paragraph.
   This is the second sentence in the paragraph. </p>
   ```

   The `style` attribute is used to add styles to an element, such as color, font, size, and more.

   ```
   \texttt{<p style="color:red;">This is a red paragraph.</p>}
   ```

   The `title` attribute defines some extra information about an element. The value of the title attribute will be displayed as a tooltip when you mouse over the element:

   ```
   <p title="I'm a tooltip">This is a paragraph.</p>
   ```

3. **HTML links**:

   ```
   <a href="https://www.w3schools.com">This is a link</a>
   ```

   In here, the `href` attribute specifies the URL of the page the link goes to.

4. **HTML images**: The source file (`src`), alternative text (`alt`), `width`, and `height` are provided as attributes:

   ```
   <img src="w3schools.jpg" alt="W3Schools.com" width="104"
                                   height="142">
   ```

   The `src` attribute specifies the path to the image to be displayed. There are two ways to specify the URL in the `scr` attribute:

   (a) **Absolute URL**: Links to an external image that is hosted on another website. For example,

   ```
   src="https://www.w3schools.com/images/img\_girl.jpg"
   ```

   Note that external images may be under copyright, and you cannot control external images; it can suddenly be removed or changed.

   (b) **Relative URL**: Links to an image that is hosted within the website. Here, the URL does not include the domain name. If the URL begins without a slash, it will be relative to the current page. If the URL begins with a slash, it will be relative to the domain.

   ```
   src="img\_girl.jpg"          src="/images/img\_girl.jpg"
   ```

It is recommended to use relative URLs.

The `width` and `height` attributes specifies the width and height of the image in pixels. The `alt` attribute for the `<img>` tag specifies an alternate text for an image, if the image for some reason cannot be displayed.

```
<img src="img\_girl.jpg" alt="Girl with a jacket">
```

5. **HTML breaks**: This is the same as if we pressed `enter`. This actually does not require an end tag, making it an **empty tag**.

```
<p> This is a <br> paragraph with a line break. </p>
```

Note that these tags are **not case-sensitive** (e.g. `<p>` is the same as `<P>`). It is recommend to always quote the attribute values.

Viewing HTML Source

In order to *view the HTML source code* for a webpage, right-click on the HTML page and select "View/Show Page Source". This will open a window containing the HTML source code of the page.

In order to *inspect an HTML element*, right-click on an element (or a blank area) and choose "Inspect Element" to see what elements it is made up of (shown in both HTML and CSS). You can also edit the HTML or CSS on the fly in the Elements or Styles panel that opens.

9.2 HTML Display

Note that with HTML, you cannot change the display by adding extra spaces or extra lines in your HTML code. The browser will automatically remove any extra spaces and lines when the page is displayed.

```
<p>
This paragraph
contains a lot of lines
in the source code,
but the browser
ignores it.
</p>

<p>
This paragraph
contains         a lot of spaces
in the source         code,
but the        browser
ignores it.
</p>
```

Definition 9.2.1 (HTML Horizontal Rules)**.** The `<hr>` tag defines a thematic break in an HTML page, and is most often displayed as a horizontal line. It is used to separate content in an HTML page.

```
<h1>This is heading 1</h1>
<p>This is some text.</p>
<hr>
<h2>This is heading 2</h2>
<p>This is some other text.</p>
<hr>
```

The `<hr>` tag is an empty tag, which means that it has no end tag.

HTML Preformatted Text

The HTML `<pre>` element defines preformatted text. The text inside a `<pre>` element is displayed in a fixed-width font (usually Courier) and it preserves both spaces and line breaks:

```
<pre>
  My Bonnie lies over the ocean.

  My Bonnie lies over the sea.

  My Bonnie lies over the ocean.

  Oh, bring back my Bonnie to me.
</pre>
```

9.3 HTML Styles and Text Formatting

The `style` attribute is used to add styles to an element, such as color, font, size, and more. The HTML `style` attribute has the following syntax:

```
<tagname style="property:value;">
```

The *property* is a CSS property. The *value* is a CSS value.

1. The CSS `background-color` property defines the background color for an HTML element. For example, the following sets the background color for a page to powderblue:

   ```
       <body style="background-color:powderblue;">

       <h1>This is a heading</h1>
       <p>This is a paragraph.</p>

       </body>
   ```

   We can set background colors for two different elements:

   ```
       <body>

       <h1 style="background-color:powderblue;">This is a heading</h1>
   ```

```
    <p style="background-color:tomato;">This is a paragraph.</p>

    </body>
```

2. The CSS `color` property defines the text color for an HTML element:

```
    <h1 style="color:blue;">This is a heading</h1>
    <p style="color:red;">This is a paragraph.</p>
```

3. The CSS `font-family` property defines the font to be used for an HTML element:

```
    <h1 style="font-family:verdana;">This is a heading</h1>
    <p style="font-family:courier;">This is a paragraph.</p>
```

4. The CSS `font-size` property defines the text size for an HTML element:

```
    <h1 style="font-size:300\%;">This is a heading</h1>
    <p style="font-size:160\%;">This is a paragraph.</p>
```

5. The CSS `text-align` property defines the horizontal text alignment for an HTML element:

```
    <h1 style="text-align:center;">Centered Heading</h1>
    <p style="text-align:center;">Centered paragraph.</p>
```

HTML contains several elements for defining text with a special meaning.

1. HTML `<b>` and `<strong>` element.

```
    <b>This text is bold</b>
    <strong>This text is important!</strong>
```

2. HTML `<i>` and `<em>` element. A screen reader will pronounce the words in `<em>` with emphasis.

```
    <i>This text is italic</i>
    <em>This text is emphasized</em>
```

3. HTML `<small>` element defines smaller text:

```
    <small>This is some smaller text.</small>
```

4. HTML `<mark>` element defines marked or highlighted text:

```
    <p>Do not forget to buy <mark>milk</mark> today.</p>
```

5. HTML `<del>` element defines text that has been deleted (browsers usually strike a line through deleted text):

```
    <p>My favorite color is <del>blue</del> red.</p>
```

6. HTML <ins> element defines text that has been inserted into a document (browsers usually underline inserted text):

```
<p>My favorite color is <del>blue</del> <ins>red</ins>.</p>
```

7. HTML <sub> and <sup> elements define subscript and superscript text.

```
<p>This is <sub>subscripted</sub> text.</p>
<p>This is <sup>superscripted</sup> text.</p>
```

Definition 9.3.1 (HTML Quotation and Citation Elements)**.** Listed.

1. HTML <blockquote> element defines a section that is quoted from another source. Browsers indent <blockquote> elements:

```
<p>Here is a quote from WWF's website:</p>
<blockquote cite="http://www.worldwildlife.org/who/index.html">
For 50 years, WWF has been protecting the future of nature.
The world's leading conservation organization,
WWF works in 100 countries and is supported by
1.2 million members in the United States and
close to 5 million globally.
</blockquote>
```

2. HTML <q> tag defines a short quotation. Browsers insert quotation marks around quotations.

```
<p>WWF's goal is to: <q>Build a future where people live in
                                          harmony with nature.</q></p>
```

3. HTML <abbr> defines an abbreviation or an acronym, which can give useful information to browsers, translation systems, and search engines. It is recommended to use the global title attribute the show the description for the abbreviation/acronym when you mouse over the element.

```
<p>The <abbr title="World Health Organization">WHO</abbr> was
                                          founded in 1948.</p>
```

4. HTML <address> defines the contact information for the author/owner of an article. It can be an email address, URL, physical address, phone number, social media, etc. The text in the <address> element usually renders in *italic* and browsers will always add a line break before and after the <address> element.

```
<address>
Written by John Doe.<br>
Visit us at:<br>
Example.com<br>
Box 564, Disneyland<br>
USA
</address>
```

5. HTML `<cite>` tag defines the title of a creative work (e.g. a book, poem, song, movie, painting, etc. It is usually rendered in **italic**.

```
<p><cite>The Scream</cite> by Edvard Munch. Painted in 1893.</p>
```

6. HTML `<bdo>` (bi-directional override) tag is used to override the current text direction.

```
<bdo dir="rtl">This text will be written from right to left</bdo>
```

Definition 9.3.2 (Comments)**.** Comments can be added to the HTML source by using the following syntax:

```
<!-- Write your comments here -->
<!-- Do not display this image at the moment
<img border="0" src="pic_trulli.jpg" alt="Trulli">
-->
```

Note that there is an exclamation point (!) in the start tag, but not in the end tag.

HTML Links

HTML links are hyperlinks that you can click on to jump to another document. When you move the mouse over a link, the mouse arrow will turn into a little hand.

The HTML `<a>` tag defines a hyperlink. It has the following syntax:

```
<a href="url">link text</a>
<a href="https://www.w3schools.com/">Visit W3Schools.com!</a>
```

By default, links will appear as follows in all browsers:

1. An unvisited link is underlined and blue.
2. A visited link is underlined and purple.
3. An active link is underlined and red.

Note that links can be styled with CSS to get another look.

Furthermore, by default, the linked page will be displayed in the current browser window. To change this, you must specify another **target** for the link. The `target` attribute can have one of the following values:

1. `_self` - Default. Opens the document in the same window/tab as it was clicked.
2. `_blank` - Opens the document in a new window or tab.
3. `_parent` - Opens the document in the parent frame.
4. `_top` - Opens the document in the full body of the window.

For example, we can use `target="_blank"` to open the linked document in a new browser window or tab:

```
<a href="https://www.w3schools.com/" target="_blank">Visit W3Schools!</a>
```

Unlike for absolute URLs, a local link (a link to a page within the same website) is specified with a *relative URL* (without the `https://www` part)

```
<h2>Absolute URLs</h2>
<p><a href="https://www.w3.org/">W3C</a></p>
<p><a href="https://www.google.com/">Google</a></p>

<h2>Relative URLs</h2>
<p><a href="html_images.asp">HTML Images</a></p>
<p><a href="/css/default.asp">CSS Tutorial</a></p>
```

We can use other HTML elements as links:

1. To use an image as a link, just put the `<img>` tag inside the `<a>` tag:

```
    <a href="default.asp">
    <img src="smiley.gif" alt="HTML tutorial"
                                                style="width:42px;height:42px;">
    </a>
```

2. Use `mailto:` inside the `href` attribute to create a link that opens the user's email program.

```
    <a href="mailto:someone@example.com">Send email</a>
```

3. To use an HTML button as a link, you have to add some JavaScript code. JavaScript allows you to specify what happens at certain times, such as a click of a button:

```
    <button onclick="document.location='default.asp'">HTML
                                                Tutorial</button>
```

Creating Bookmarks in HTML

To create a bookmark, you first create the bookmark and then add the link to it. When the link is clicked, the page will scroll down or up to the location with the bookmark.

1. First, use the `id` attribute to create a bookmark.

```
    <h2 id="C4">Chapter 4</h2>
```

2. Then, add a link to the bookmark ("Jump to Chapter 4"), from within the same page:

```
    <a href="#C4">Jump to Chapter 4</a>
```

3. You can also add a link to a bookmark on another page:

```
    <a href="html_demo.html#C4">Jump to Chapter 4</a>
```

Chapter 10

Python 3

10.1 Regular Expressions

A **RegEx**, or **Regular Expression**, is a sequence of characters that forms a search pattern. RegEx can be used to check if a string contains the specified search pattern. It uses the `re` module.

10.2 Web Scraping

The relevant packages that must be imported are `bs4, urllib.request`.

We first define the URL of the website that we want to scrape. For example, let us take the wikipedia article on the Eastern Front of WWII.

```python
from bs4 import BeautifulSoup
from urllib.request import Request, urlopen
import urllib.request

#Define the URL we want to scrape
wiki_url = r"https://en.wikipedia.org/wiki/Eastern_Front_(World_War_II)"
```

Then, we use the `request` function in `urllib.request` (which is a library for opening URLs) to make a **HTTP request** to the website. More information can be found in the HTTP section in this book, but in general once the Python program identifies the IP address of the host computer hosting the requested URL, it sends the HTTP request (usually `HTTP/1.1`). The host computer then sends back an HTTP response with both the content and metadata about it. We usually set this HTTP response as a variable:

```python
>>>response_object = urllib.request.urlopen(wiki_url)
>>>print(type(response_object))
>>>print(response_object)
<class 'http.client.HTTPResponse'>
<http.client.HTTPResponse object at 0x7fa3382de280>
```

Definition 10.2.1. The `urlopen` function has the following paramaters:

```
urllib.request.urlopen(url, data=None, timeout, cafile=None, capath=None,
                                        cadefault=False, context=None)
```

1. Open the URL `url`, which can either be a string or a `Request` object.
2. `data` must be an object specifying additional data to be sent to the server, or `None` if no such data is needed.
3. The optional `timeout` parameter specifies a timeout in seconds for blocking operations like the connection attempt. Default is the global default timeout setting.
4. The optional `cafile, capath` parameters specify a set of trusted CA certificates for HTTPS requests.
5. `cadefault` parameter is ignored.
6. `context` can be set to `None`.

With this, HTML request object, we can input it into the `BeautifulSoup` function of the `bs4` library, which gives us a `BeautifulSoup` object that represents the document as a nested data structure (note that this is not a string!).

```
soup = BeautifulSoup(response_object, "html.parser")
print(type(soup))     #<class 'bs4.BeautifulSoup'>
print(soup)
#This gives us a nice representation of the HTML as a nested data
                                        structure.
```

One may notice the second argument `"html.parser"`; we can leave this alone. HTML parsing is basically taking in HTML code and extracting relevant information like the title of the page, paragraphs in the page, headings in the page, links, bold text, etc. This is really what `BeautifulSoup` does.

To make the returned BeautifulSoup object a bit easier to read, we can use `.prettify()`.

```
html_doc =
"""<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and
                                        their names were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
"""
soup2 = BeautifulSoup(html_doc, 'html.parser')
print(soup2.prettify())
# <html>
#  <head>
#   <title>
#    The Dormouse's story
#   </title>
#  </head>
#  <body>
#    <p class="title">
#     <b>
#      The Dormouse's story
#     </b>
#    </p>
#    <p class="story">
#     Once upon a time there were three little sisters; and their names were
#     <a class="sister" href="http://example.com/elsie" id="link1">
#      Elsie
#     </a>
#     ,
#     <a class="sister" href="http://example.com/lacie" id="link2">
#      Lacie
#     </a>
#     and
#     <a class="sister" href="http://example.com/tillie" id="link3">
#      Tillie
#     </a>
#     ; and they lived at the bottom of a well.
#    </p>
#    <p class="story">
#     ...
#    </p>
#   </body>
# </html>
```

Definition 10.2.2 (Find method)**.** The `find` method finds the first instance of a HTML element with a certain tag.

```
    soup = BeautifulSoup(response_object, "html.parser")
    print(soup.find("a"))
    #<a id="top"></a>
```

We can confirm that this is indeed the first element with tag `<a>` when looking at the page source in the browser.

1. If we would like it to have an additional attribute of a `href` (which means that it has a link attached), then we can just write

```
    link = soup.find("a", href=True)
    print(link)
    #<a class="mw-jump-link" href="#mw-head">Jump to navigation</a>
    #Note that this is not a string, even though it looks like a
                                        string.
    print(type(link))      #What is the type? It is a tag.
    #bs4.element.Tag
    print(link.name)       #What is the name of this tag?
    #'a'
```

2. We can look for other attributes in the `<a att1 = ' ' att2 = ' ' >` and use them to find the elements with tag `tag` that have these attributes.

```
    link = soup.find("tag", att1=True, att2 = "This")
    print(link)
```

3. We can also get the dictionary of all the attributes with the `.attrs` method.

```
print(link.attrs)
#{'class': ['mw-jump-link'], 'href': '#mw-head'}
```

4. The `text` attribute returns the text (i.e. the text that is actually shown on the website) of the element with the tag.

Definition 10.2.3 (Find_all method)**.** The `find_all` method works exactly like the `find` method, but it displays a `ResultSet` object (which acts like a list) with all the elements having the required tag and meeting certain attribute requirements.

```
links = soup.find_all('a', href=True)
print(type(links))      #type of this find_all object
print(len(links))
#bs4.element.ResultSet
#3098                   #This website contains 3098 links!
```

1. We can loop through this `ResultSet` as if it were a list, but we can't assume that every element of this set is going to have the same attributes. For example, every element has a `href` attribute (since we built the list that way), but not every element has the `title` attribute. Therefore, we must use `try, except`.

```
for link in links[0:10]:
    print(link['href'])
    try:
        print(link['title'])
    except:
        continue
```

2. We can `find_all` elements that have either one of multiple tags by inputting a list rather than a string as the argument:

```
#search for tables and a tags
tables_and_a = soup.find_all(['table', 'a'])
```

3. We can add additional arguments to make the attributes more specific:

```
#table headers with style attribute
table_headers = soup.find_all('th', style='width: 10\%;')
print(table_headers)
#[<th style="width: 10\%;">Date </th>]

#find all the wiki tables
wiki_tables = soup.find_all('table', class_='wikitable')
```

Since `class` is already a keyword in Python, the attribute is `class_`.

4. We can `find_all` with multiple attributes, by inputting a dictionary.

```
#a list of all tags that have a href and a title, and have the
                                class value mw-direct.
```

```
    a_tags_multi = soup.find_all('a', {'href':True, 'title':True,
                                         'class':"mu=redirect"})
```

5. We can also define a function to find the items

```
    def list_with_links(tag):
        return tag.name == 'li' and len(tag.find_all('a'))>7

    #list items with a tags a
    list_with_a = soup.find_all(list_with_links)
```

Family Tree

Definition 10.2.4 (Children and Descendants)**.** Given a HTML code, a **child** of a tag is the direct subtag, and the **descendants** of a tag are all the children and children of those children, etc.

```
#define the simple tree
simple_tree =
'''<html><body><a><b>text1</b><c>text2</c></a></body></html>'''

#pass the simple tree into our parser to create some simple soup.
simple_soup = BeautifulSoup(simple_tree, 'html.parser')

#we can always print it in a familiar structure.
print(simple_soup.prettify())

#<html>
# <body>
#  <a>
#   <b>
#    text1
#   </b>
#   <c>
#    text2
#   </c>
#  </a>
# </body>
#</html>
```

If we want just a list of the children we can use the content attribute:

```
a_content = simple_soup.a.contents  #the 'a' stands for the a tag
display(a_content)
[<b>text1</b>, <c>text2</c>]  #this is actually not a list!

#an identical way to get the list of children is as such:
print(simple_soup.a.children) #this is a list!
```

All of the descendants of a tag can be gotten as such:

```
print(simple_soup.a.descendants)
[<b>text1</b>, text1, <c>text2</c>, text2]
```

Definition 10.2.5 (Parents)**.** We can go backwards to get the parent tag:

```
#get the parent of the 'a' tag
print(simple_soup.a.parent)
#<body><a><b>text1</b><c>text2</c></a></body>

#get the parent of the parent of the 'a' tag
print(simple_soup.a.parent.parent)
#<html><body><a><b>text1</b><c>text2</c></a></body></html>
```

Now, just like the descendants, we can get all the parents of the 'a' tag.

```
for parent in simple_soup.a.parents:
    print(parent)
#<body><a><b>text1</b><c>text2</c></a></body>
#<html><body><a><b>text1</b><c>text2</c></a></body></html>
#<html><body><a><b>text1</b><c>text2</c></a></body></html>
```

The first line is the body tag, the second is the html tag, and the final line is the entire document itself. This can also be found with the `find_parent` method:

```
print(simple_soup.b.find_parent())   #find parent of b tag
print(simple_soup.b.find_parents())  #find all parents of b tag
#<a><b>text1</b><c>text2</c></a>
#[<a><b>text1</b><c>text2</c></a>,
#<body><a><b>text1</b><c>text2</c></a></body>,
#<html><body><a><b>text1</b><c>text2</c></a></body></html>,
#<html><body><a><b>text1</b><c>text2</c></a></body></html>]
```

We've learned how to go up or down, but we can do sideways to get a tag's **siblings**.

Definition 10.2.6. A tag can have either a sibling after it or before it, denoted by the `next_sibling` and `previous_sibling` method.

```
#print the element with tag b
print(simple_soup.b)
#<b>text1</b>

#print the next sibling (what comes after) of b
print(simple_soup.b.next_sibling)
#<c>text2</c>

#print the previous sibling of b
print(simple_soup.b.previous_sibling)
#None
```

We can see that b has one sibling after it, which is c, but since it is the first element in its generation, there is no previous sibling. This can also be done with the `find_next_sibling` method.

```
simple_soup.b.find_next_sibling()      #find the next sibling of b tag
simple_soup.b.find_next_siblings()     #find all next siblings of b tag
simple_soup.b.find_previous_sibling()  #find the previous sibling of b tag
simple_soup.b.find_previous_siblings() #find all previous siblings of b tag
```

Definition 10.2.7 (Order of Parsing using next_element)**.** The order in which a HTML file was parsed can be determined by the `next_element` method.

```
#grab the body
print(simple_soup.body)
#<body><a><b>text1</b><c>text2</c></a></body>

print(simple_soup.body.next_element)
#<a><b>text1</b><c>text2</c></a>
```

However, the order in which it is parsed does not align always with the sibling order. Therefore, the `next_element` and `previous_element` method simply refers to the order in which the HTML was parsed, nothing more. This can also be found using the `find_next` method.

```
simple_soup.a.find_next()        #find the next element of a tag
simple_soup.a.find_all_next()    #find all next elements of a tag
simple_soup.a.find_previous()    #find the previous element of a tag
simple_soup.a.find_all_previous()#find all previous elements of a tag
```

Adding Attributes to Certain Tags

Say that you have parsed a HTML file and you have the, say, first a-tag element. We can display its `href` attribute normally as such:

```
a_tag = table.a
print(a_tag)
#<a href="#cite_note-37">[37]</a>

print(a_tag['href'])
# '#cite_note-37'
```

We can also *add our own attribute* `style` *to this tag* as such:

```
a_tag['style'] = 'my new width'
print(a_tag)
#<a href="#cite_note-37" style="my new width">[37]</a>
```

This is extremely useful since it allows us to *tag* certain elements with an attribute for easy identification of these elements for later on when we're analyzing data.

We can also *add text/string to the tag.*

```
#here we add a new string to the a tag
a_tag.string = "My New String"
print(a_tag)
#<a href="#cite_note-37" style="my new width">My New String</a>
```

Since we can add values, we also have the capability to delete them. The `clear` method clear all the string in the tag.

```
#grab the first a tag
tag = table.a
```

```
print(tag)
#<a href="#cite_note-37" style="my new width">My New String</a>

tag.clear()
print(tag)
#<a href="#cite_note-37" style="my new width"></a>
```

Note that all the changes that you do to the tags are permanent! You must re-request the HTML and parse through it again to get the original version!

Definition 10.2.8 (Extracting a tag)**.** Given a certain section of HTML code, we can choose to **extract** a certain portion of the code that has a specific tag, which basically removes that portion from the original code. We can take this extracted portion and store it as a variable.

```
#grab a table body
print(table.tbody)

#extrac the first table header
th_tag = table.tbody.tr.extract()

#display the extracted tag
print(th_tag)
#<th style="width: 10\%;">Date</th>
```

Again, note that every time you extract a portion of the HTML code, you're permanently removing it! After extracting enough times, there won't be any more tags to extract and you'll get an error statement (which is a drawback).

Definition 10.2.9. To get the strings that belong to a certain tag, we do the following:

```
#printing the first string that you find in the table header tag
print(table.th.string)

#get all the strings that belong to a table body
for string in list(table.tbody.strings)[0:10]:
    print(string)
```

We have to convert `table.tbody.strings` to a list in order to slice it (since we only want to show the first 10 strings).

However, this will produce a lot of line breaks, so the more popular one to go to is the `stripped_strings` method.

```
for string in list(table.tbody.stripped_strings)[0:10]:
    print(string)
```