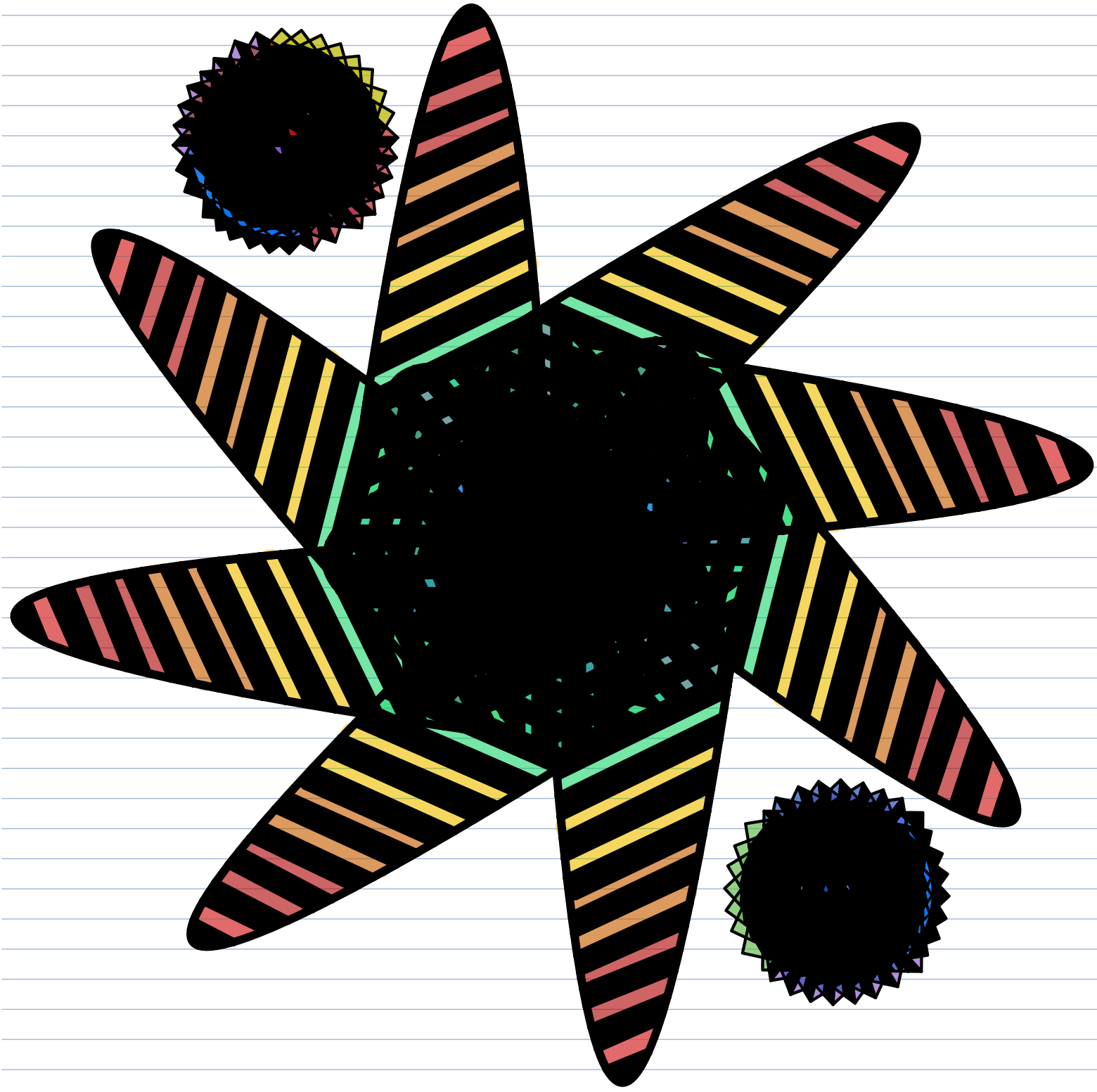
CS 181 Introduction to Theoretical Computer Science



There are two aspects of computing:

- Data representations of objects → Representations → "Data"
- Algorithms: operations on data → Operations → "Algorithms"

Representing Objects:

If we can represent objects of type T, then we can represent lists/tuples of objects of type T.

Define Representation: $E : \Theta \to \{0,1\}^*$ where E is an encoding function that is one-to-one

Aside on notation: kleene star all permutations of a set

set A: $\{0,1\} = [0], [1]$

set A^2: $\{0,1\}^2 = [00], [01], [10], [11]$

$A^* = \emptyset \cup A \cup A^2 \cup A^3 \cup \dots A^k \cup \dots$

Valid Encoding: E is a valid encoding function if and only if there exists a decoding function:

$D = \{0,1\}^* \longrightarrow \Theta \; ; \; D(E(x)) = x$

Ex #1: Represent natural numbers: $E : \mathbb{N} \to \{0,1\}^*$

we can use binary: our E would be $NtoB(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ NtoB(\lfloor \frac{n}{2} \rfloor) \cdot n \bmod 2 \end{cases}$

to show that it is a valid encoding we would show that there is a valid decoding which would be the conversion from binary to an natural number. $\sum_{i=0}^{n} n(2^i)$

Ex #2: Integers

$E : \mathbb{Z} \longrightarrow \{0,1\}^* \Rightarrow E(n) = \begin{cases} 0 \circ NtoB(n) & \text{if } n \geq 0 \\ 1 \circ NtoB(-n) & \text{if } n < 0 \end{cases}$

We can apply a similar decoding to that of the NtoB

Ex #3: Rational Numbers:

1/2 → (1, 2), 2/3 → (2, 3), -3/4 → (-3, 4)

(Numerator, Denominator) ⇒ ZtoB(p) ∘ ZtoB(q) (concatonation) ⇒ however with this approach we cannot tell where the first number ends and the second begins:

(1, 3) ⇒ 111 ⇒ (3, 1)

Instead if we were to do the following: we convert every 0 to 00 and every 1 to 11. This creates a unique identifier of 01 which we can then use to indicate the end of a number.

Decoding: keep reading until there is a 01, before this is a number, after is the denominator.

Prefix free encoding:

$E: \Theta \to \langle 0,1 \rangle^*$ is prefix free if for all $x \neq y \in \Theta$, $E(x)$ is not a prefix of $E(y)$.

Ex: NtoB (binary representation)

NtoB(2) = 10

NtoB(5) = 101

Ex ℤtoB (duplicating and adding 01)

Is a prefix encoding since the 01 termination ensures that unless two numbers are the same they won't have a common prefix.

Prefix encoding is powerfull since it gives us a way to form lists of objects very easily.

Suppose we have a prefix-free encoding: $E: \Theta \to \{0,1\}^* \Rightarrow \overline{E}: \Theta^* \to \{0,1\}^*$ by $\overline{E}(x_1, x_2, \dots, x_n) = E(x_1) \circ E(x_2) \circ E(x_3) \circ \dots \circ E(x_n)$. Then $\overline{E}$ is a valid encoding of Θ^*.

Proof: Suppose someone gave the binary sequence: $\overline{E}(x_1 \dots x_n) = E(x_1) \circ E(x_2) \circ \dots \circ E(x_n)$. We can keep reading from left to right until the sequence matches an encoding. Once we find it, chop it off to recover the first object and proceed.

Theorem: Suppose $E: \Theta \to \{0,1\}^*$ is an encoding. We define pfE to stand for prefix free encoding.

Define $pfE: \Theta \to \{0,1\}^*$ as follows

$$pfE(a) = \begin{cases} \to \text{compute } E(a) \\ \to \text{replace 0 with 00 \& 1 with 11} \\ \to \text{Add 01 at the end.} \end{cases}$$

Then pfE is a prefix free encoding.

Proof: the 01 can never occurr when values are duplicated and hence only occurrs at the end in an even position.

Remarks about prefix free encoding:

The above version of prefix encoding has the number of bits doubled each time: length $= 2 \cdot |E(x)| + 2$. It therefore grows by factors of 2 $\therefore 2^n$.

However there-exist better encodings that would grow according to: $|E(x)| + (2\log_2|E(x)| + 2)$

Furthermore when we combine a prefix encoding we will loose the prefix encoding and will have to redo the process.

Represent integers as $\{0,1\}^*$
↳ Prefix-free encoding
↳ Represent list as integers
↳ Prefix-free encoding
↳ Represent list of lists
↳ These are images!

Through this process we have shown that the world is binary we can succesfully replicate this process to represent anything

*It was actually proven that we cannot have a one-to-one mapping from real numbers to $\{0,1\}^*$

Discrete Math Review:

Sets: unordered collection of objects with no repeats.

Union: $S \cup T = \{x : x \in S \text{ or } x \in T\}$

Intersection: $S \cap T = \{x : x \in S \text{ and } x \in T\}$

Set minus: $S \setminus T = \{x : x \in S \text{ and } x \notin T\}$

Compliment: $\bar{S} = \{x : x \notin S \text{ but } x \in U\}$

Cartesian product: $S \times T = \{(x,y), x \in S, y \in T\}$

$S^k = \underbrace{S \times S \times \ldots \times S}_{k\text{-times}} = \{(x_1, x_2, \ldots x_k) : x_i \in S \ \forall_i = 1, \ldots, k\}$

Kleene Star: $S^* = \bigcup_{k=0}^{\infty} S^k$

where $S^0 = \{\varepsilon\} \Rightarrow$ empty string

(Has infinite elements but elements have non-infinite length)

Function: f of $S \to T$: $f(S) = \{y \in T : \exists x \in S, f(x) = y\}$

domain ↑ ↑ Codomain

One-to-One (Injective): $\forall_{x_1, x_2 \in S} : x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)$

Onto (surjective): $\forall_y \in T : \exists x \in S$ where $f(x) = y$

Bijective: if both 1 to 1 and onto.

$\exists$ 1-1 f $S \to T$ iff $|S| \leq |T|$

$\exists$ onto f $S \to T$ iff $|S| \geq |T|$

$\exists$ bijection $S \to T$ iff $|S| = |T|$

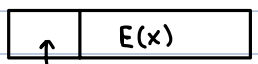
A Better PFE function

let $E : \Theta \Rightarrow \{0,1\}^*$ be an encoding

let $F : \mathbb{N} \to \{0,1\}^*$ be a pfE such that $\forall_n \in \mathbb{N}$, $|F(n)| = 2\log(n) + 2$.

(Duplicate each bit; for 01)

We define pfE: $\Theta \to \{0,1\}^*$ as $pfE = F(|E(x)|) \circ E(x)$

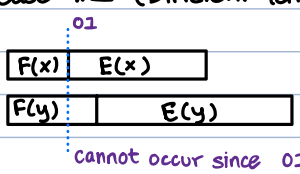


↑ Contains length of E(x) ended by a 01

Claim: $pfE(x) = F(|E(x)|) \circ E(x)$ is prefix free; The proof:

Take any $x \neq y \in \Theta$. For the sake of contradiction assume $pfE(x)$ is $pfE(y)$:

Case #1 (Different length for x, y)



$F(|E(x)|) \neq F(|E(y)|) \Rightarrow |E(x)| \neq |E(y)|$ which means that $F(|E(y)|)$ contains a 01 in the middle which is a contradiction to our definition of F

cannot occur since 01 cannot appear is F(y) since F is pre-fix free

Case #2 (Equal Length)

Then $F(|E(x)|) = F(|E(y)|) \Rightarrow |E(x)| = |E(y)|$ since the strings are same length:

$\Rightarrow E(x) = E(y)$ and we know $E(x) \neq E(y)$ and this is a contradiction.

Algorithms:

A series of steps to solve the same problem

- Formally put: transform inputs to desired output

Specification: Function $f: \{0,1\}^* \longrightarrow \{0,1\}^*$

Steps are defined to be some set of basic operations

Inputs	Out
⋮	⋮

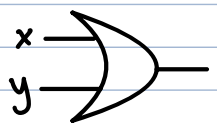
One way to think of this, is a generalized truth table: where a function maps a set of inputs to some outputs.

Boolean Circuits:

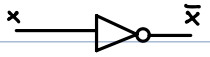
We compute with AND/OR/NOT are the basic operations

AND "$\wedge$": $\{0,1\}^2 \to \{0,1\}$ $\Rightarrow$ $AND(a,b) = \begin{cases} 1 \text{ if } a=b=1 \\ 0 \text{ else} \end{cases}$

OR "$\vee$": $\{0,1\}^2 \to \{0,1\}$ $\Rightarrow$ $OR(a,b) = \begin{cases} 0 \text{ if } a=b=0 \\ 1 \text{ else} \end{cases}$



NOT "$\neg, \sim$": $\{0,1\} \to \{0,1\}$ $\Rightarrow$ $NOT(a) = \begin{cases} 1 \text{ if } a=0 \\ 0 \text{ if } a=1 \end{cases}$



Example: Majority three elements. $MAJ3: \{0,1\}^3 \longrightarrow \{0,1\}$

$$MAJ3(a,b,c) = \begin{cases} 1 \text{ if } a+b+c \geq 2 \\ 0 \text{ else} \end{cases}$$

$$MAJ3 = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$$

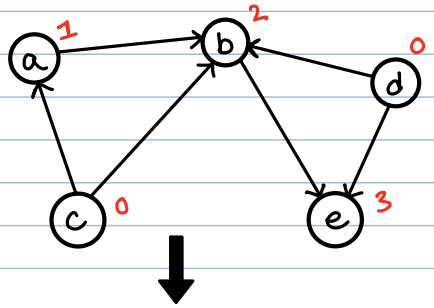
Example XOR:

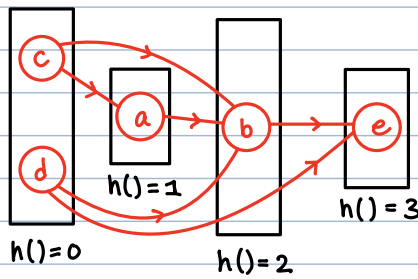
$XOR2(a,b) = [A \wedge (\neg B)] \vee [(\neg A) \wedge B]$ where $XOR2: \{0,1\}^2 \longrightarrow \{0,1\}$

$XOR3(a,b,c) = XOR2(XOR2(a,b), c)$ where $XOR3: \{0,1\}^3 \to \{0,1\}$

"Solve the problem" ≡ Means compute the function

Directed Acyclic Graph (DAG) and Boolean Circuits





If we draw a boolean circuit we see that it acts like a DAG

Defining a boolean circuit: $A(n, m, s)$ — n: # variables, m: # outputs, s: size

It is a DAG with $n+s$ vertices

- Exactly n of these vertices are labeled as input: $x[0] \ldots x[n-1]$
- The other s vertices are gates.
- Each and has two inputs and one output, each OR has two inputs and one output. Each not gate has one input and one output.
- m of the gates are labaled as outputs: $y[0], y[1] \ldots, y[m-1]$

Topological Sorting:

If $G = (V, E)$ is a DAG, there is a "layering $h: V \to \mathbb{N}$ such that for every edge (i,j), $h(i) < h(j)$

From Circuits to Computation:

Given we have a boolean circuit with n inputs, m outputs, and s gates:

1. Layer the DAG using topological sort so that all input verticies are in layer zero
2. We have computed in layers $0, 1, \cdots, k-1$:
3. For each vertex in layer k:
   - If AND gate take and
   - If OR gate take or
   - If NOT gate take not

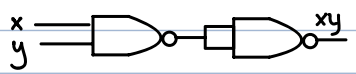
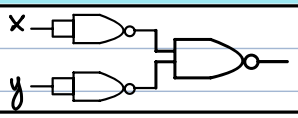
Output: $\{0,1\}^n \longrightarrow \{0,1\}^m$

We say a circuit C computes a function f is $f(x) = C(x)$ for all $x \in \{0,1\}^n$

NAND Circuits:

Same definition of boolean circuit but the only gates allowed are NAND $\Rightarrow$ (NOT(AND (A,B)))
we have the same computing function $C: \{0,1\}^n \rightarrow \{0,1\}^m$.

NAND is a universal gate we can represent and, or, and not using NAND.

Not gate:	AND	OR
		

Both NAND and Boolean circuit computational power are equivilent: if f is a function that is computable by a boolean circuit, if and only if a function that is computable by a NAND circuit. $\Longrightarrow$ equivilence

---

Theorem: every function $f: \{0,1\}^n \longrightarrow \{0,1\}^m$ can be computed by a Boolean circuit of size $O(n \cdot m \cdot 2^n)$.

Proof $(m=1)$. $f: \{0,1\}^n \rightarrow \{0,1\}$ equivilent to a truth table of 2^n rows.

let us pickout the rows where our values are 1: $S = \{\alpha : f(\alpha) = 1\}$ (Binary string.)

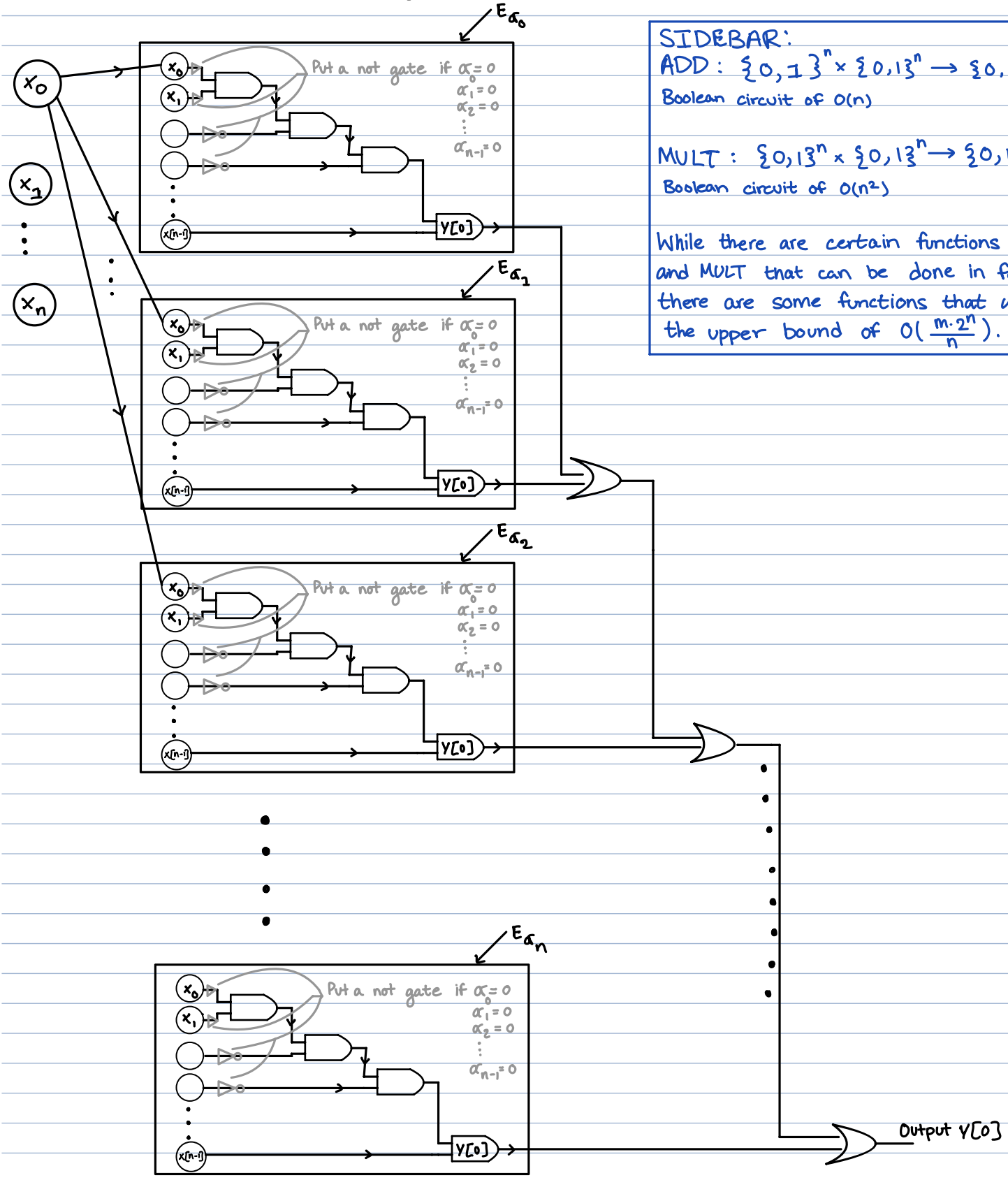
For each string $\alpha \in \{0,1\}^n$, define $E_\alpha : \{0,1\}^n \longrightarrow \{0,1\} \Rightarrow E_\alpha(x) = \begin{cases} 1 & \text{if } x = \alpha \\ 0 & \text{otherwise} \end{cases}$

$\Rightarrow E_{(1,1,1,\cdots,1)}(x) = \text{AND}(x[0], x[1], \ldots, x[n-1]) \Rightarrow$ # gates $= 2N-1$
for any bit in alpha that is not a 1 we can not that value. Hence we know that we can represent our E function using circuit elements.
Part #2: We can use E_α to "compute" arbitrary functions of f: $S = \{\alpha : f(\alpha) = 1\} = \{\alpha_0, \alpha_1, \cdots \alpha_{N-1}\}$
this is equivilent to checking if x equals any of the alpha values. If x equals even one of the α_s we output 1.

$$f(x) = \text{OR}(E_{\alpha_0}(x), E_{\alpha_1}(x), \ldots E_{\alpha_{N-1}}(x))$$

This would take $(\#\, E_\alpha \text{ gates})(\alpha) \Rightarrow (2n-1)(m) + (m-1) \Rightarrow (2n-1)(2^n) + (2^n - 1) \Rightarrow O(n \cdot 2^n)$
Remark # of output bits is 1. If not we would repeat for each $m \Rightarrow O(n \cdot m \cdot 2^n)$
Note: here we are taking the OR and AND of multiple bits it is important to mention that this can be accomplished by repeated chaining. This is how we get our boolean circuit.

Visualization of the theorem: Any function can be represented a boolean function.



SIDEBAR:

ADD : $\{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^{n+1}$

Boolean circuit of $O(n)$

MULT : $\{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^{2n}$

Boolean circuit of $O(n^2)$

While there are certain functions like ADD and MULT that can be done in fewer gates there are some functions that will require the upper bound of $O(\frac{m \cdot 2^n}{n})$.

# of gates used: number of gates used for $E_\alpha()$ times the number of we need to compute E_α. It takes $2n-1$ gates to do E_α worst case. We would have to do maximum of 2^n $E_\alpha()$ calculations if every truth table value is 1.

Hence: $2^n(2n-1) + 2^n - 1 \Rightarrow n \cdot 2^n \Rightarrow$ for where $m \neq 1$ we get $O(m \cdot n \cdot 2^n)$

Remark: $\exists$ a circuit of size $O(\frac{m \cdot 2^n}{n})$

Viewing Circuits as $\{0,1\}^*$ strings:

We will view circuits as inputs.

"General Purpose" Computers: computers that are not specific to any task, using this idea.

Represent Circuit as string

- Some functions require exponential size
- Universal Circuit

Thm: Every NAND(n,m,s) circuit can be represented by a Boolean string of length of $O(S \cdot \log(n+s))$

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

$S(s) \leq C \cdot (n+s)\log_2(n+s)$

*Talks about circuits of atmost S gates

Looking at the encoding:

- ⟶ specify inputs (n)
- ⟶ specify output (m)
- ⟶ # gates ($S_0 \leq S$)
- ⟶ Which nodes correspond to output
- ⟶ Links between gates. (inputs to each gate)

$(n, m, S, \underbrace{y[0], y[1], \dots, y[m-1]}_{\text{nodes of outputs}}, \overbrace{\text{Connections}}^{\text{Total } S_0})$

range: $[n, n+S_0]$

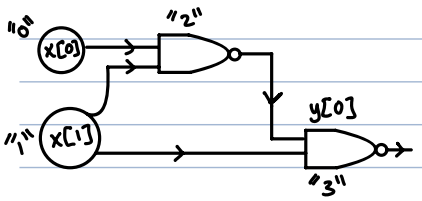
Each gate has exactly two inputs, so we can represent incoming edges for a node as a tuple.

Ex: (,) (,), ..., (,)

- (,): nodes into gate 1.
- (,): Nodes into gate S_0

This provides a numerical encoding of a circuit

Ex:



(2, 1, 2, "3", (0,1), (2,1))

Step #2: go from representation to $\{0,1\}^*$

m will tell us how many bits to read

$(n, m, S, \underbrace{y[0], y[1], \dots, y[m-1]}_{}, \underbrace{\text{Connections}}_{})$ ⇒ $E: \text{Circuits} \to \{0,1\}^*$

We can take prefix free encoding of integers ⇒ $\in \{0,1\}^*$

We have an agreed upon format and we know the size, n, m, so we can just translate to binary.

Step #3: Big O

Recall that PFE of an integer takes $\leq 2 \cdot \log_2(a)$ bits).

(n, m, S, y[0], y[1], ..., y[m-1], Connections) ⇒ we will make everything into a PFE (not nesseceray after n,m,s)

$2\log_2(n) + 2\log_2(m) + 2\log_2(S_0) + 2m \cdot \log_2(n+S_0) + 4S_0 \cdot \log_2(n+S_0)$ lets assume: $(m \leq n+S_0)$

$\leq 2\log_2(n+S_0) + 2\log_2(n+S_0) + 2\log_2(n+S_0) + 2m \cdot \log_2(n+S_0) + 4S_0 \cdot \log_2(n+S_0)$

$= \log_2(n+S_0)[6 + 2m + 4S_0] \leq \log_2(n+S_0)[6 + 2(n+S_0) + 4(n+S_0)] = (6(n+S_0)+6)\log_2(n+S_0)$

$\leq 12(n+S_0) \cdot \log_2(n+S_0) \leq 12(n+S)\log_2(n+S)$ ⇒ circuits with n variables, m outputs, size $\leq S$ can always be represented in $O(12(n+S)\log_2(n+S))$

Functions require exponential-size circuits:

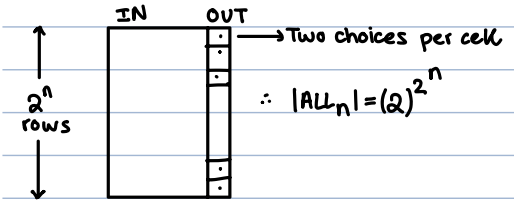
Thm: there exist functions: $f:\{0,1\}^n \to \{0,1\}$ that require circuits of size $\frac{c \cdot 2^n}{n}$ where $c>0$ a universal constant.

$ALL_n = \{ f : \{0,1\}^n \to \{0,1\} \}$ ⇒ class of all functions

$SIZE_n(s) = \{$ All functions with 1-bit output $\}$ ⇒ That are computable by circuits of size $\leq s$

Idea: $|ALL_n| > |Size_n(\frac{c \cdot 2^n}{n})|$ for $(c = \frac{1}{100})$

How to count ALL_n?



$\therefore |ALL_n| = (2)^{2^n}$

How to count $Size_n(s)$?

$|Size_n(s)| \leq$ Number of circuits on n inputs, 1 output, at most s gates

length $\leq 12(n+s)\cdot \log_2(n+s)$ (Idea: counting using encodings)

$|Size_n(s)| \leq$ # strings of $\{0,1\}$ with length $\leq 12(n+s)\log(n+s)$

* Number of strings $\in \{0,1\}^*$ of length $\leq \ell$
$= 2^0 + 2^1 + 2^2 + \dots + 2^\ell = 2^{\ell+1}$

$\therefore |Size_n(s)| \leq 2(2)^{12(n+s)\log_2(n+s)}$

Comparing ALL_N and $Size_n(S)$:

↳ is $|All_n| \geq |Size_n(s)| \Rightarrow (2)^{2^n} \geq 2(2)^{12(n+s)\log_2(n+s)}$

* The number of functions computable with s gates is smaller than the total number of functions.

Therefore there exists some function on n bits that requires $\frac{2^n}{24 \cdot n}$ gates. ⇒ $\Omega(\frac{2^n}{24n})$ (Gates) for some functions

---

Universal Circuits:

Eval: $\{0,1\}^{S(n,m,s)} \times \{0,1\}^n \to \{0,1\}^m$ where $EVAL(C(x)) = C(x)$.

$$Eval(C(x)) = \begin{cases} C(x) & \text{if } C \text{ is a valid circuit} \\ 0^m & \text{if } C \text{ is invalid} \end{cases}$$

Circuit C → Binary String $\{0,1\}$ → Eval → Output of C(x)

There exists a circuit that computes EVAL with inputs of $S(n,m,s)+n$ with a max size of $O(S^2 \cdot \log S)$. $(S \geq n, m)$.

Summary of circuits:

- → Circuits can be implemented on physical devices
- → Every function can be computed by circuits
- → Some functions require exponential size.
- → Universal Circuits: ∃ a circuit of size $s^2 \log(s)$ that can simulate all size S circuits.

Aside on Quantum Computing:
the closest way to model quantum computing is through boolean circuits. Instead our basic operations change from logic gates to quantum gates.

Physical Extended Church-Turing Thesis:

a function $f:\{0,1\}^n \to \{0,1\}$ can be computed by using s physical resources, then it can be computed by circuits that use roughly s gates. So far this thesis has not been proven wrong. It has stood the test of time. For every example we must calculate or look for a hidden cost that is associated with it.

Proving that XOR is not universal:

XOR(a,b) ⇒

a	b	XOR(a,b)
0	0	0
0	1	1
1	0	1
1	1	0

⇒ $(a+b) \bmod 2$ ⇒ Prove $\exists S \subseteq \{1,\dots,n\}$ such that $C(x) = \sum_{i \in S} x_i \pmod 2$

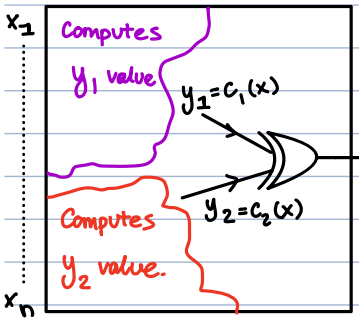
↳ There is a subset

Proof by Induction on s (# gates):

Base Case: $s=1$ $C(x) = XOR(x_1, x_2)$, $S = \{x_1, x_2\}$ (if $x_1 \neq x_2$)

or $XOR(x_1, x_2)$, $S = \emptyset$ if $x_1 = x_2$

Inductive Hypothesis: statement is true for all circuits of size $\leq s-1$



Observe that y_1, y_2 are computed of circuits of size $\leq s-1$

⇒ By induction we know that there are sets S_1, S_2 such that

$y_1 = \sum_{i \in S_1} x_i \pmod 2$ and $y_2 = \sum_{i \in S_2} x_i \pmod 2$

⇒ $C(x) = [C_1(x) + C_2(x)] \bmod 2 \Rightarrow \sum_{i \in S} x_i \pmod 2$ where $S = S_1 \Delta S_2$ $(S_1 \Delta S_2 = (S_1 \setminus S_2) \cup (S_2 \setminus S_1))$

⇒ hence by induction we prove that $C(x)$ always follows $\sum_{i \in S} x_i \bmod 2$. To prove that the function is not universal we provide an example that does not follow that function.

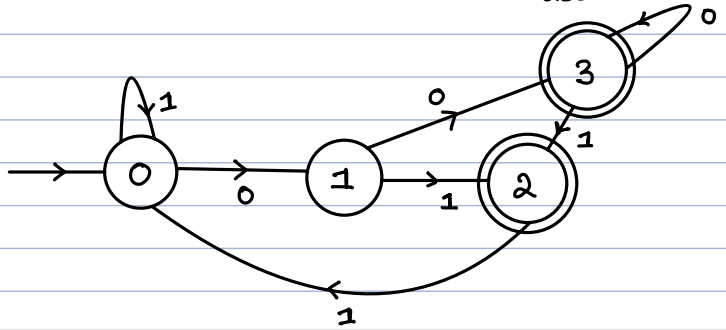
Majority with NANDs efficiently:

Prove that there is some constant c such that for every $n>1$, there is a NAND circuit of at most $c \cdot n$ gates that computes the majority function on n input bits: $MAJ_n: \{0,1\}^n \to \{0,1\}$. We define $MAJ_n(x) = 1$ iff $\sum_{i=0}^{n-1} x_i > n/2$.

Writing a recurrence: $MAJ_N = 2MAJ(\frac{N}{2}) +$

DFA to see if second to last bit is zero:

$f(x): \{0,1\}^* \to \{0,1\}$ ⇒ $f(x) = \begin{cases} 1 & \text{if } 2^{nd} \text{ to last bit is zero} \\ 0 & \text{else} \end{cases}$



1011

1001

Prove that every finite $L \subseteq \{0,1\}^k$ has a DFA that could compute it.

We can represent every string in $\{0,1\}^k$ in a binary tree, where each node in the tree is a state. we know we then have reached the end of a string if it is a leaf. For all leaf nodes if the path to that node is an string in L then mark that leaf as an accepting state.

Functions with Unbounded length:

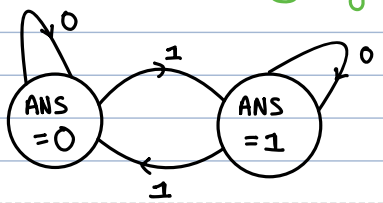
$f: \{0,1\}^* \to \{0,1\}$, circuits are great for unbounded length but how do we do computation on unbounded length.

Ex: $XOR(x) = \begin{cases} 1 & \text{if number of 1's is odd} \\ 0 & \text{otherwise} \end{cases}$

```
def XOR(x):
    ans = 0
    for i in range(len(x)):
        ans = (ans + x[i]) % 2
    return ans;
```

Algorithm:
→ Until I scan end of the input
→ Read one bit of input
→ Update my "state"
→ Output result of my "state"

These algorithms are called single pass constant memory algorithm.



An algorithm is a finite answer to an infinite number of questions.

Formally these are studied under "formal languages"; Language just means a set of strings that satisfy something

---

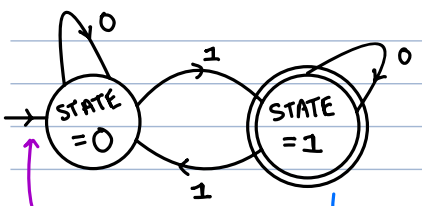
Deterministic Finite Automaton (DFA) ⇒ (Not universal)

- A DFA is described by: C states over $\{0,1\}$ which is a pair, (T, S) where $T: [C] \times \{0,1\} \to [C]$ and $S \subseteq [C]$ (S is a subset of all possible states. represents sucess)
  ↳ $[C] = \{0, 1, \ldots, C-1\}$
  $T(i, a) = j$ (Shows the change from a state i to a state j based on the input a.
  $\in \{0, \ldots, C-1\} \cdot \{0,1\}$ $\quad \{0, 1, \ldots, C-1\}$.

- A DFA defines a function $D: \{0,1\}^* \to \{0,1\}$ as follows:
  a. Start with a state $S_0 = 0$
  b. For $i = 0, 1, \ldots, len(x)-1$:
     $S_{i+1} = T(S_i, x[i])$
  c. Output 1 if last state $\in S$ otherwise 0.

With DFA we dont care about input size, since we can keep switching states. $T[C] \times \{0,1\} \to [C]$ is the transition function.

We say that a DFA $D = (T, S)$ computes a function $f: \{0,1\}^* \to \{0,1\}$ if $f(x) = D(x)$ $\forall$ $x \leftarrow \{0,1\}^*$



STATE	BIT	STATE	
0	0	0	T(0,0) = 0
0	1	1	T(0,1) = 0
1	0	1	T(1,0) = 1
1	1	0	T(1,1) = 1

DFA ≡ Finite State Machine
$D(x) = XOR(x)$ for all x.
"Alphabet" ≡ $\{0,1\}$ (Σ)

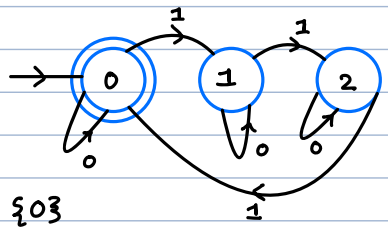
Dangling arrow to show starting state

A double circle indicates elements of S ∴ $S = \{1\}$.

Example DFA:

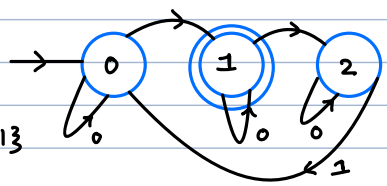
T ≡

State	Bit	State
0	0	0
0	1	1
1	0	1
1	1	2
2	0	2
2	1	0



$S = \{0\}$

$D(x) = \begin{cases} 1 & \text{if number of 1's in x is divisible by 3} \\ 0 & \text{otherwise.} \end{cases}$

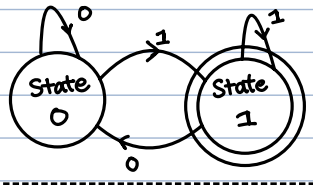


$S = \{1\}$

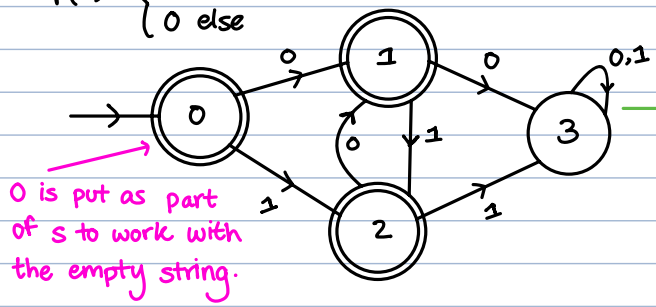
$D(x) = \begin{cases} 1 & \text{if number of 1's in x is 1 mod 3} \\ 0 & \text{otherwise.} \end{cases}$

More Examples of DFAs

$f(x) = \begin{cases} 1 \text{ if } x \text{ ends in } 1 \\ 0 \text{ else} \end{cases} \Rightarrow$



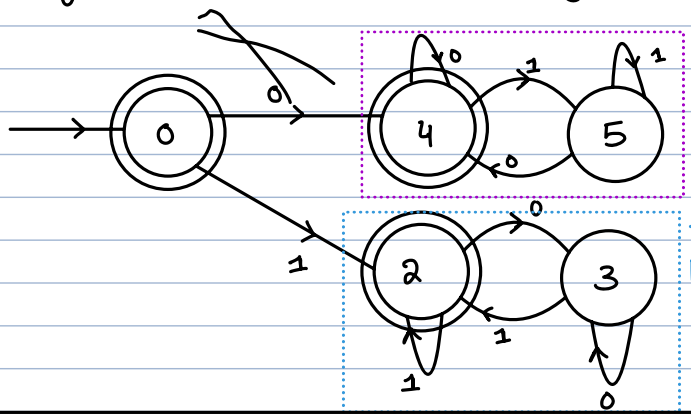
$f(x) = \begin{cases} 1 \text{ if } x \text{ has alternating 0's, 1's} \\ 0 \text{ else} \end{cases}$



→ State 3 is a "dead state" once we enter we cannot leave.

0 is put as part of s to work with the empty string.

Design a DFA that outputs 1 on strings that have the same starting and ending bits.



→ Purple box checks if last bit is 0

→ Blue box checks if last bit is 1.

Note how we use two states to check what the first bit is.

Anatomy of a DFA:

fixed or bounded:

- → Number of states
- → Transition table T
- → Set of accepting states S

Input length is unbounded.

(Q, Σ, T, S_0, S)

- Q: Set of States: $\{0, 1, \ldots, C-1\}$
- Σ: Alphabet → {0,1}
- T: Transition function
- S_0: Starting state
- S: Output states.

Languages:

$f: \langle 0,1 \rangle^* \rightarrow \{0,1\}$; $L_f = \{x : f(x) = 1\}$

→ DFA computes function f.

→ DFA computes Language L (ie; $D(x) = 1 \Leftrightarrow x \in L$

≡ DFA "recognizes" language L.

Performing Operations on DFA's:

Suppose $f_1 : \{0,1\}^* \rightarrow \{0,1\}$ is computed by DFA D_1

Suppose $f_2 : \{0,1\}^* \rightarrow \{0,1\}$ is computed by DFA D_2

let $f(x) = f_1(x) \wedge f_2(x)$

```
def ANDfg(x):
    a = f(x)
    b = f(x)
    return a∧b
```

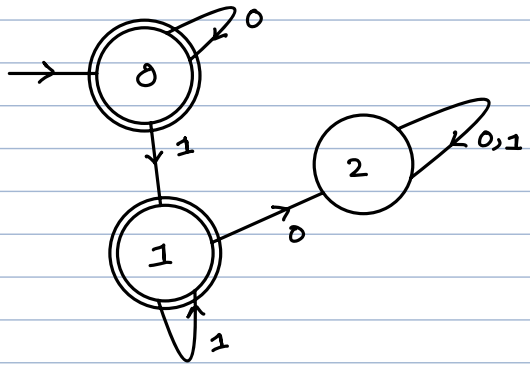
This is not single pass!

How can we compute such operations?

DFA Design Practice Problems:

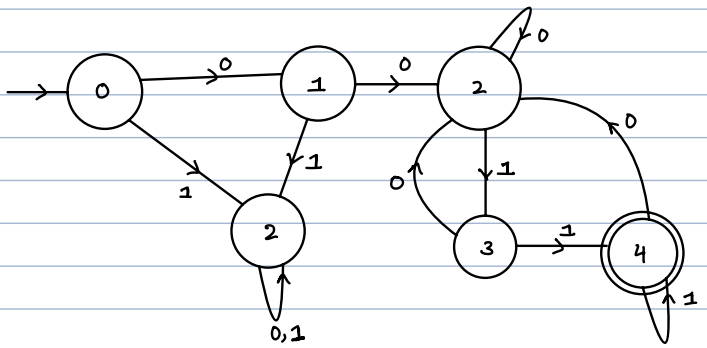
Ex #1)

Design a DFA that excepts $x \in \{0,1\}^*$ iff $x = a0b$ where $a \in \{0\}^*$, $b \in \{1\}^*$



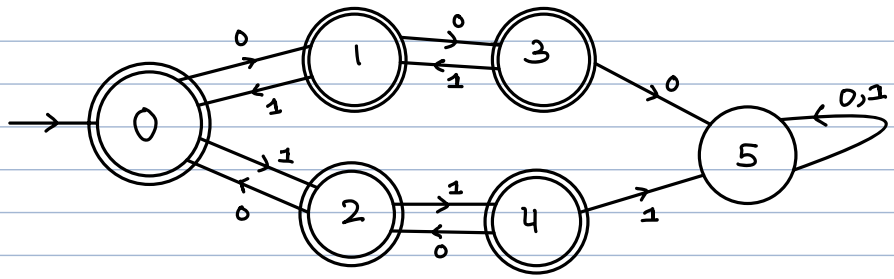
Ex #2)

Design a DFA that accepts x iff x begins with 00 and ends with 11



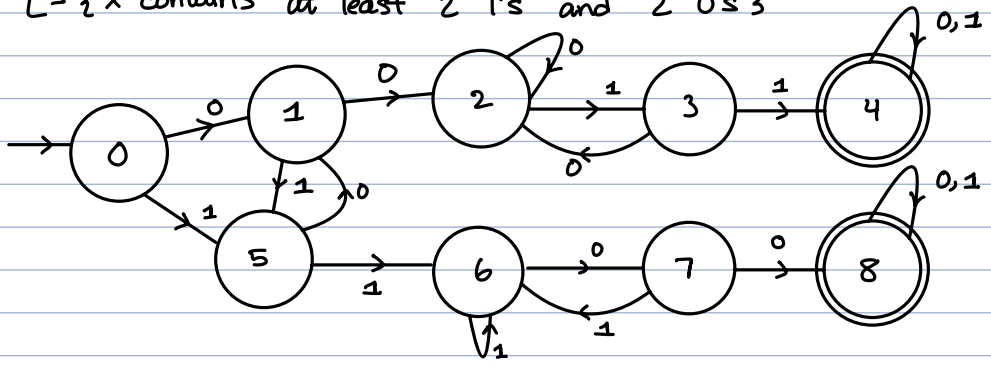
Ex #3:

Design a DFA that accepts $x \in \{0,1\}^*$ iff $\forall$ prefix $\bar{x}$ of x, if $|(\#1\text{'s in } \bar{x}) - (\#0\text{'s in } \bar{x})| \leq 2$



Ex #4:

L = { x contains at least 2 1's and 2 0's }



AND using DFA's

DFA D_1 that computes f_1: C_1 states: (T_1, S_1)
DFA D_2 that computes f_2: C_2 states: (T_2, S_2)
$\Rightarrow f_1 \wedge f_2 \Rightarrow$ Run D_1 and D_2 in parallel

$C = C_1 \times C_2$ where (i,j) where $i \in \{0,1,\dots,C_1-1\}$ $\Rightarrow$ $T((i,j),a) = (T_1(i,a), T_2(j,a))$
where $j \in \{0,1,\dots,C_2-1\}$ $S = \{(i,j) : i \in S_1, j \in S_2\}$

Thm: DFAs are closed under "ANDs" f_1, f_2 computable by DFAs $\Rightarrow f_1 \wedge f_2$ computable by DFAs.
aka L_1, L_2 are recognized by DFAs ... Then there is a DFA that recognizes $L_1 \cap L_2$

OR using DFA's

DFA D_1 that computes f_1: C_1 states: (T_1, S_1)
DFA D_2 that computes f_2: C_2 states: (T_2, S_2)
$\Rightarrow f_1 \vee f_2 \Rightarrow$ Run D_1 and D_2 in parallel

$C = C_1 \times C_2$ where (i,j) where $i \in \{0,1,\dots,C_1-1\}$ $\Rightarrow$ $T((i,j),a) = (T_1(i,a), T_2(j,a))$
where $j \in \{0,1,\dots,C_2-1\}$ $S = \{(i,j) : i \in S_1 \text{ or } j \in S_2\}$

Thm: DFAs are closed under "OR" f_1, f_2 computable by DFAs $\Rightarrow f_1 \vee f_2$ computable by DFAs.
aka L_1, L_2 are recognized by DFAs ... Then there is a DFA that recognizes $L_1 \cup L_2$

NOT using DFA's

DFA D_1 that computes f_1: C_1 states: (T_1, S_1) is $\overline{f_1}$ $\{0,1\}^* \to \{0,1\}$ computable by f.
Transition functions stay the same; $\overline{D} = (T, \{0,1,\dots,C-1\} \setminus S)$
↳ Complement of S.

Thm: DFAs are closed under "NOT" f_1 computable by DFAs $\Rightarrow \overline{f_1}$ computable by DFAs.
aka L_1 are recognized by DFAs ... Then there is a DFA that recognizes $\overline{L_1}$

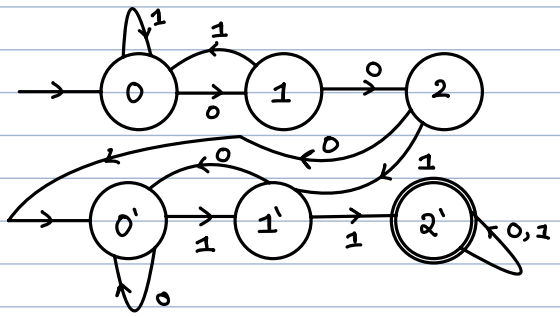
CONCATENATION:

f, g are functions: $\{0,1\}^* \to \{0,1\}$. The concatenation of f, g: $f \circ g : \{0,1\}^* \to \{0,1\}$.
$f \circ g(x) = 1$ if and only if we can split x in some way where $x = [x_1][x_2]$, $f(x_1) = 1$, $g(x_2) = 1$.

Languages: $L_f, L_g \Rightarrow L_{f \circ g} = L_f \circ L_g = \{x : x = x_1 \cdot x_2 \text{ where } x_1 \in L_f, x_2 \in L_g\}$

A motivating example:

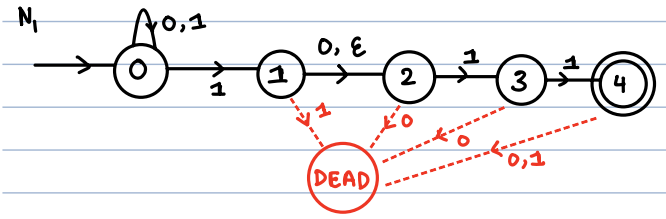
$f(x) = 1$ if x has 00 in it
$g(x) = 1$ if x has 11 in it



In this example we show that we connect the final output of f(x) to "input" of g(x). However this would not work if we had multiple output states or if the output state did something useful

There is a universal way to do concatenation (not the method above) which we will explore further.

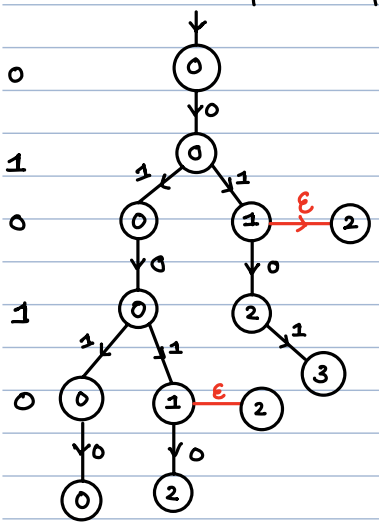
NON-Deterministic Finite Automaton (NFA) ⇒ (Not universal)



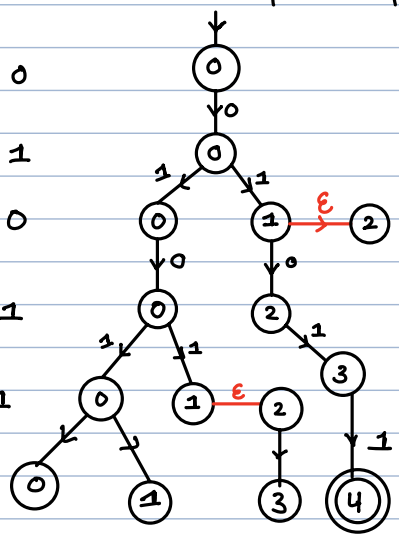
If we are missing an arrow it means that it goes to a dead state.
In the actual NFA drawing the diagram would not contain the parts in red.

NFA:
- → can have multiple outgoing edges with same label out of a state.
- → edges that are missing go to a dead state
- → Some edges are labeled by "ε" (without reading any bit we move to the next state)
- → We output 1 if any branch has a state with S after the last bit is read.

Ex what does N_1 do on input for 01010:



Ex what does N_1 do on input for 01011:



This can be thought of as running the NFA with different paths at the same time. Similar to branching into a new universe at the same time.

Formal Definition of NFA:

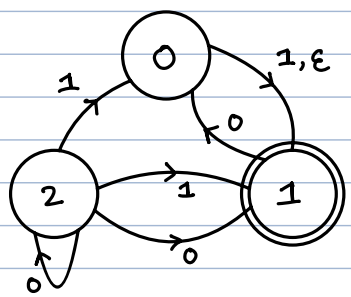
$N = (T, S)$ where $T : [C] \times \{0, 1, \varepsilon\} \longrightarrow \text{Power}([C])$ and $S \subseteq [C]$

↳ $\text{Power}([C]) = \{I : I \subseteq [C]\}$ all subsets of C.

$N(x) = 1$ iff one of the final states is $\in S$; otherwise it is zero

---

Ex: Writing the transition function for a NFA



	0	1	ε
1	∅	{1}	{1}
2	{0}	∅	∅
3	{1, 2}	{0, 1}	∅

Why use NFA's.

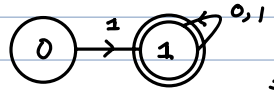
Suppose we have a function f_1: computable by a DFA D_1 and a function f_2 computable by a DFA D_2. Can we construct an NFA for $L_1 \circ L_2$? Yes!

For all accepting states in f_1 we add ε transitions from first machines accepting states to the start of the second machine. Hence N computes f_1 cat f_2.

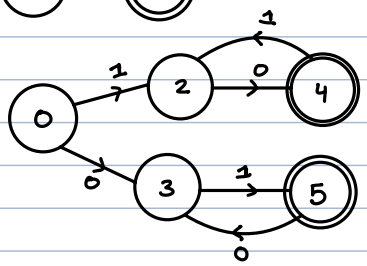
Ex: Concatonation using a NFA:

L_1: {starts with a 1} ; L_2: {alternates between zero and one}

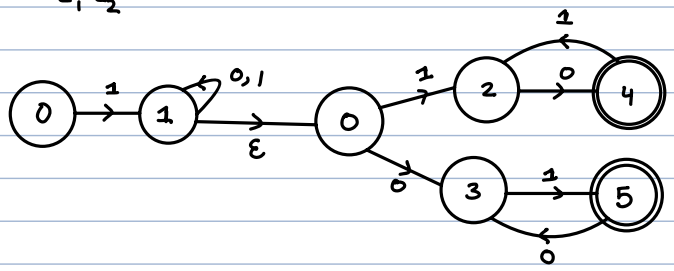
L_1:



L_2:



Then: $L_1 \circ L_2$



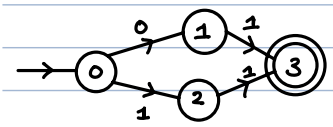
KLEENE Star Operation on functions:

if x can be broken up as $x^0 \circ x^1 \circ x^2 \circ \dots \circ x^{l-1}$ such that $f(x^0)=1, f(x^1)=1, f(x^2)=1 \dots f(x^{l-1})=1$.
by definition f^*(empty string) = 1.
if f is computable by a DFA. Then f^* is computable by a NFA.

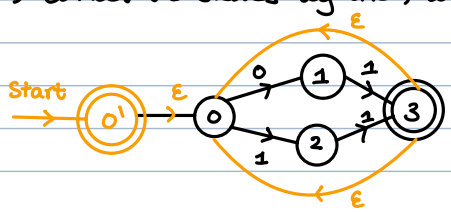
↳ Add "ε" transitions from accepting states to the start state. ⇒ this works for everything but the empty string. to solve this we just add an initial dummy state to absorb the empty string.

Ex: Write the NFA for $\{01, 11\}^*$

1). Write DFA for {01, 11}

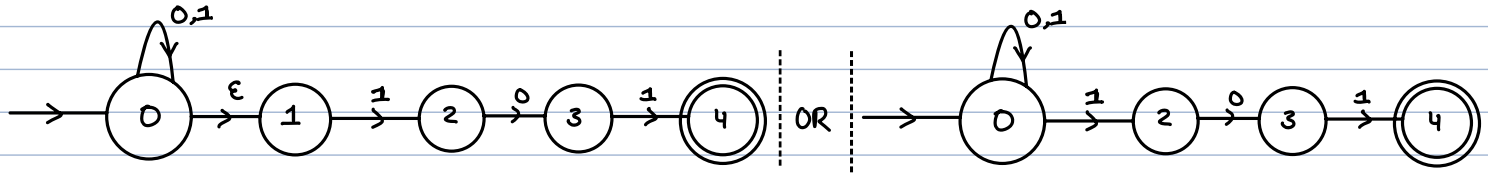


2). Connect the states together, to form the NFA:

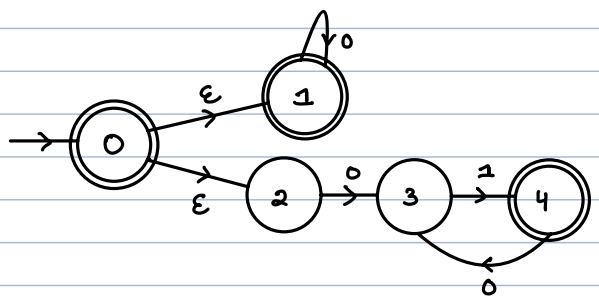


Examples of NFAs:

Ex #1: L= {x: x ends with 01}



Ex #2: $L = \{x: \{0\}^* \cup \{01\}^*\}$

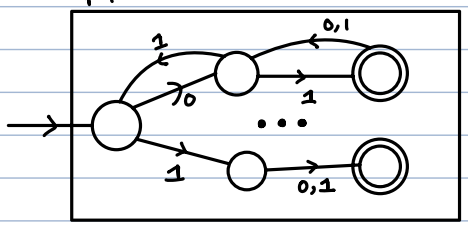


A Deep Dive into $f^{Reverse}$

$f^{reverse}(x) = f(Reverse(x))$

↳ x written backwards

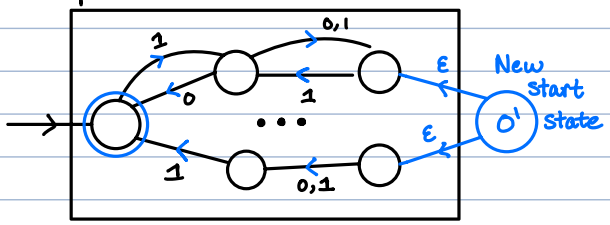
f :



If f(x) has a DFA then we can construct a NFA for $f^{Reverse}$

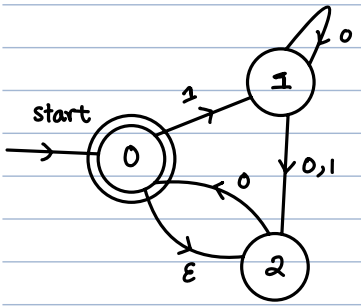
The big idea is that we follow the path backwards:

$f^{reverse}$



1). Add a new start state
2). Add ε transitions from new start state to all previous accepting accepting states.
3). Reverse all arrow directions
4). Make old start state new accept state
5). Old accepting states no longer accept.

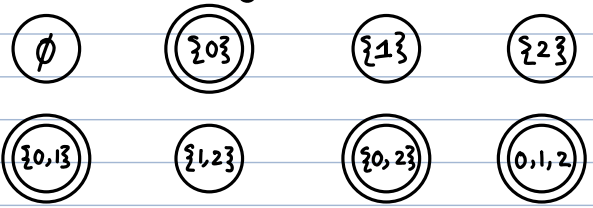
Ex of Converting Between a NFA to DFA :



⇒

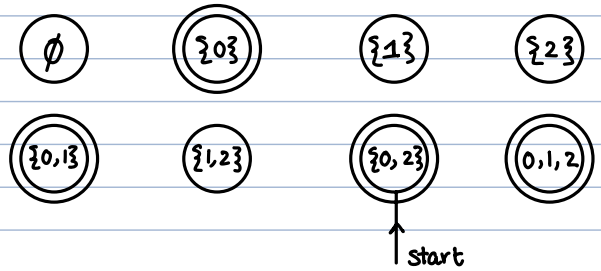
① First draw the power set : $\emptyset$ $\{0\}$ $\{1\}$ $\{2\}$ $\{0,1\}$ $\{1,2\}$ $\{0,2\}$ $\{0,1,2\}$

② We make our accepting states anything that contains the zero.

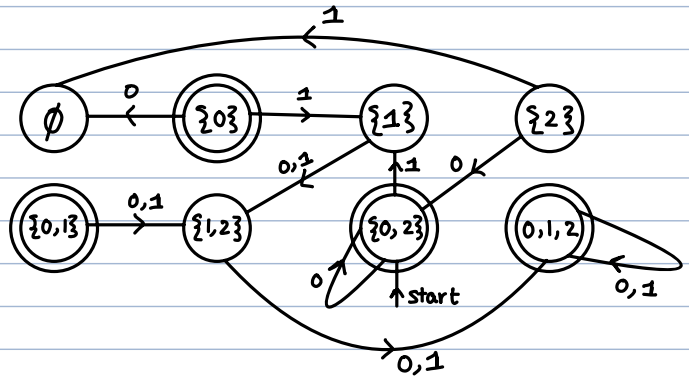


* Proof that this conversion is possible is done on the next page.

③ Our start state becomes the ε transition of our original start state: Eps $\{0\} \Rightarrow \{0,2\}$



④ Add in all of the edges of corresponding transition function.



Thm: Every NFA has and equivilent DFA: For every NFA N, $\exists$ a DFA D such that $N(x) = D(x)$ $\forall$ x.

Corollary #1: $f_1, f_2 : \{0,1\}^* \to \{0,1\}$ are computable by a DFA then $f_1 \circ f_2$ is computable by a DFA.
Corollary #2: $f_1 : \{0,1\}^* \to \{0,1\}$ is also computable by a DFA.

PROVING THAT every NFA has a DFA:

Proving that every NFA has a DFA will have two parts. First where we do not have ε transitions and then we will add the ε transitions back.

Idea #1: We can merge branches in an NFA if there is a duplicate node. At every level we only need to consider what states we reach, not how many such states are there. Two copies of same state at SAME Level can be collapsed into one. For each level we need to know what states are reachable at that level.

PART #1: Assume that the NFA has no ε transitions.

Suppose we have an NFA $N \equiv (T_n, S_N)$ without ε transitions; and states $[C]$

Our DFA only needs to rember the states at each level. It is like rembering subsets of $[C]$
States in the DFA = # of subsets of C = Power($[C]$) = 2^C states, where each state corresponds to a different subset of $[C]$.

Transition of DFA: T_D: $T_D(I, a) = \bigcup_{i \in I} T_N(i, a)$; This is the idea of seeing the set of *all* states reachable in the next level.
($\text{Power}([C]) \times \{0,1\} \to \text{Power}([C])$)
$I \subseteq C$
↳ Union over *all* elements in I.

New Accept States: S_D: $S_D \{ R \subseteq \{0,1,\dots,C-1\} : R \cap S_n \neq \emptyset \}$ ⇒ *all* subsets which contain at least one of the original accepting states of N.

Start State: $\{0\}$

PART #2: Adding ε transitions:

Similary to how we were storing a list of *all* reacheable states at each level all we would need to do is add where our ε transitions lead.

For a state $i \in \{0,1,\dots,C-1\}$ we say Eps(i) = all states reachable from i by only taking ε transitions + $\{i\}$

For a set $I \subseteq [C]$, $\text{Eps}(I) = \bigcup_{i \in I} \text{Eps}(i)$

Hence our new transition function becomes: $T_D = \text{Eps}\left(\bigcup_{i \in I} T_N(i, a)\right)$; and our new start state $\equiv \text{Eps}(\{0\})$
↳ Our accepting state function is the same.

Remarks:
- If the NFA had C states, then the DFA has 2^C states. There are NFA's on C states whose computation by DFAs needs 2^{C-1} states.

Remark on Closure Properties:

- We can do the closure properties on NFA's with the same closure properties as a DFA.
- However, take NOT we cannot simply invert our NFA's states. Instead we first convert our NFA to a DFA and then apply this closure property of DFA's.

Regular Expressions:

Helps with pattern matching: input: x,p ; Output: does p occurr in x?

Knuth-Morris-Pratt (KMP) Algorithm:

solves string matching in time $O(n+m)$

→ Given p, first find a DFA D on m+1 states
→ For any string x, p occurs in x if and only if $D(x)=1$ — $O(m)$ time
→ Now, mimic behaviour of D on x — $O(n)$ time

Fragile X syndrome:

Given a DNA sequence $\{A,G,C,T\}^*$ can we determine whether there exists some pattern x within that sequence which would be an indicator of some syndrome.

Regular Expression Cases:

Base Cases:
- "0" is a regex
- "1" is a regex
- Empty String: $\emptyset$ is a regex
- Empty String: ε is a regex

Compound Cases:
- → $r_1 r_2$ is a regex
- → r_1^* is a regex
- → $r_1 | r_2$ is a regex "r_1 or r_2"

Matching Regular Expressions:

→ x matches $r_1 r_2$ if $x = x_1 \circ x_2$ where x_1 matches r_1 and x_2 matches r_2.
→ x matches r_1^* if $x = x_1 \cdot x_2 \cdots x_i$ such that each x_i matches r.
→ x matches $(r_1 | r_2)$ if x matches r_1 or if x matches r_2.

ε matches ε
$\emptyset$ is not matched by anything

Regular Expressions ≡ Languages ≡ DFAs

$f_r : \{0,1\}^* \to \{0,1\}$ if $f_r(x) = \begin{cases} 1 \text{ if } x \text{ matches } r \\ 0 \text{ else} \end{cases}$
$L_r = \{ x \in \{0,1\}^* : x \text{ matches } r \}$

Theorem (Kleene):
a. For every regex r, there is a DFA D such that $f_r(x) = D(x)\ \forall x$.
b. For every DFA D, there is a regex r such that $D(x) = f_r(x)\ \forall x$.

A function f is regular if there is a regex that computes it. A language L is regular if there is a regex that recognizes it.

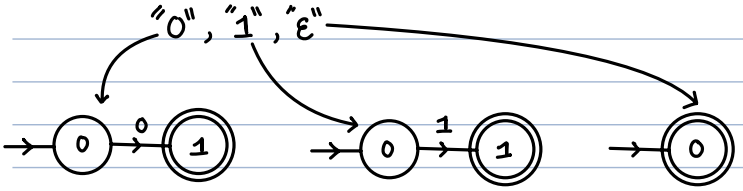
We don't have to be limited to just $\{0,1\}$; ex: (a-z|0-9)*@___.com → valid email!

We can solve it in time $O(n \cdot m)$ or $O(n + 2^{O(m)})$. GREP converts the regex to an equivalent NFA → O(m) → then simulates the NFA on input x → $O(m \cdot n)$ time.

Constructing Regex from NFAs:

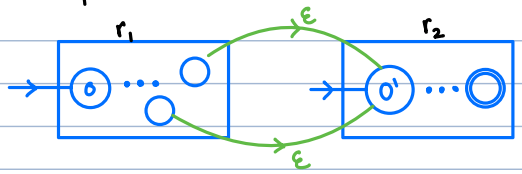
→ We built up a regex based up from an inductive standpoint

→ Base Case:



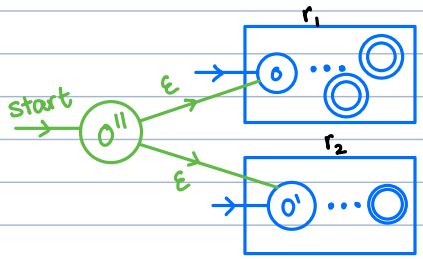
→ Compound Cases:

Suppose we have $r = r_1 \cdot r_2$ and we know r_1 and r_2 are computable by NFAs we saw that concatonation was possible.

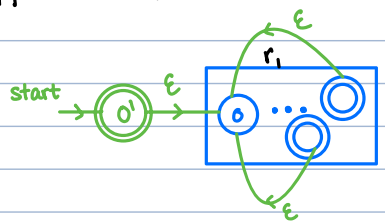


⇒ Yes!

Suppose we now need to build a $r = (r_1 | r_2)$ given NFAs for r_1, r_2



→ Suppose we have an NFA for r_1 can we do it for $(r_1)^*$

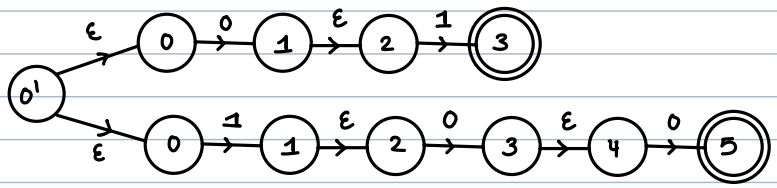


Therefore to create an NFA for a regular expression we would repeatedly follow the above steps.

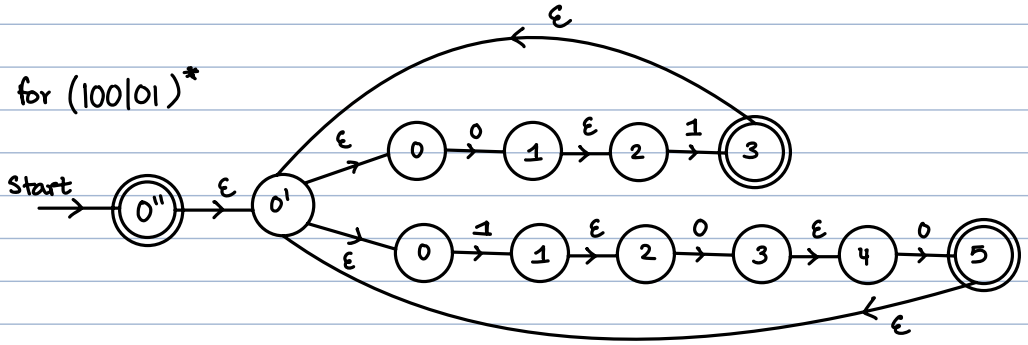
Remark: the entire process can be represented in linear time.

Ex: $r_1 = 01(100|01)^*$

NFA for 100|01:



NFA for $(100|01)^*$



And we can continue this process...

A Theoretical View on DFAs:

→ DFA, NFA, RE are a simple model that works for unbounded length inputs

→ Non-determinism → creates very powerful modularity.

→ If an NFA has m states and n length string it takes $O(m \cdot n^2)$ time.

Computational limit of DFA's:

→ DFA is single pass with constant memory; this constant memory is an issue.

Ex: $L_1 = \{x : x$ contains equal number of 0's and 1's $\}$ or MAJ(x) are NOT computable by DFAs.

Ex#2: $L_2 = \{x : x$ contains equal number of 01's and 10's $\}$ is Regular. (We just check start and end bit!)

These two look syntactically the same how do we differentiate is L can be computed by a DFA?

Pumping Lemma:

If L is a regular language. Then there exists a number p such that every string $x \in L$ of length $\geq p$, can be written as $x = a \cdot b \cdot c$

ⓐ $f(a \cdot b^i \cdot c) = 1$ for all $i \geq 0$

ⓑ length of $b > 0 \Rightarrow$ b is not empty

ⓒ length of $ab \leq p$

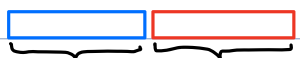
(ie: $abc \in L$ ...
$abbc \in L$ ...
$abbbc \in L$ ...)

aka ⇒ L is a regular language if ∃ a number p ∀ $x \in L$ of length at least p where the above is true.

→ we will often use this in it's contrapositive form.

Pumping Lemma MAJ element:

We define MAJ to be: $MAJ(x) = \begin{cases} 1 \text{ if at least half the bits are } 1 \\ 0 \text{ else.} \end{cases}$

how can we split up x?  ? times ? times. now let us supposse we take $x = 0^p 1^p$. Here we see that for any valve of p we choose this will result in MAJ(x) = 1.

What about b in this case? we can represent x as: $0^{p-1} 0 1^p$. Here our b value is zero. Now by the pumping lemma we get that $a \cdot b \cdot b \cdot c$ is MAJ(x)=1. However here we would get $0^{p-1} 0 0 1^p \Rightarrow 0^{p+1} 1^p$. Which means that we have more zeros than ones which results in MAJ(x)=0. This is a contradiction which means that the MAJ function is not a regular function.

Pumping Lemma Ex #1:

$L = \{1^{n!} \mid n \geq 0\}$ all strings of length 1 that are of factorial length.

let us take $x = 1^{(p-1)!} 1^p$ where $a = 1^{(p-1)!}$ and $b = 1^p$ and $c = \epsilon$. By the pumping lemma we see that $a \cdot b \cdot b \cdot c$ is part of L. However $a \cdot b \cdot b \cdot c = p!(p)$ which is not in the form of p! and hence is a contradiction. Therefore $1^{n!}$ is not a regular function.

Example #2 of Pumping Lemma:

$L = \{0^k 1 0^k : k \geq 1\}$;

Suppose L was regular then $\exists$ some number p such that pl holds:
$x = 0^p 1 0^p$. By PL we must be able to split $x = a \cdot b \cdot c$, now length$(ab) \leq p$.
$\Rightarrow$ Hence b is made up by only 0's. Then $a \cdot b \cdot b \, c$ will have more zero's before the first 1.
$a \cdot b \cdot b \cdot c$ is not in L, but PL says it should be which is a contradiction.

Example #3:

Palindrome $= \{x : x = \text{Reverse}(x)\}$

Suppose palindrome is regular. There exists some p such that PL holds:

Ex: $x = (01)^p 0 \Rightarrow a = \varepsilon, b = 01, c = (01)^{p-1} 0$ and $x = abc \Rightarrow a \cdot b \cdot b \cdot c \Rightarrow (01)(01)(01)^{p-1}(0) \Rightarrow (01)^{p+1} 0$
↳ This challenge fails since x can be pumped.

Ex#2: $x = 0^p 1 0^p \Rightarrow$ same example from example #2 which results in a contradiction and hence PALINDROME is not Regular.

Example #4:

$x = \{1^{n^2} : n \geq 1\}$ and strings with only 1's and number of 1's is a square.

Suppose L is regular. By pumping lemma we should be able to split up x. Hence $a \cdot b \cdot c$ must all be ones.
take $(a \cdot b \cdot b \cdot c) \Rightarrow$ length$(a \cdot b \cdot b \cdot c) \leq p^2 + p$ which is not a square. Hence $a \cdot b \cdot b \cdot c$ is not in the language.
this gives us a contradiction

Example #5:

PRIMENUMBER is the language $L = \{1^l \mid l \in \text{primes}\}$ the language of strings of 1s having prime length.

Suppose L (prime-number) is regular. By the pumping lemma we should be able to split up x. We know that $a \circ b \circ c$ will all be ones. Suppose we split it up as $1^r 1^s 1^t$ where $r + s + t = p$. Then we get that $1^s 1^t 1^r \in L$. However supposse we pump b p+1 times. We get $r + s(p+1) + t \Rightarrow r + s + t + s(p) \Rightarrow p + sp \Rightarrow p(1+s)$.
However $p(1+s)$ is not prime since it is divisible by p. This is a contradiction. Hence L is not regular.

Example #6:

let $L = \{0^n 1^n \mid n \geq 0\}$

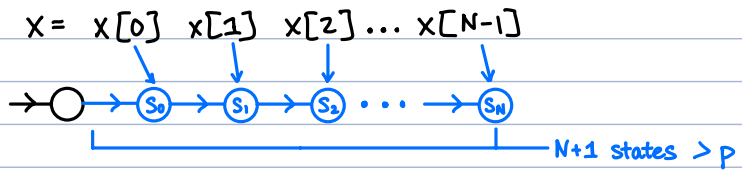
Suppose L is regular. By the pumping lemma we should be able to split up x. Take the strings $0^p 1^p$. Since we know that $|ab| \leq p$ we know that b will only have zeros. Hence if we pump multiple b values then there will be more zeros than ones which proves to be a contradiction. Hence L is not a regular language.

Remarks:

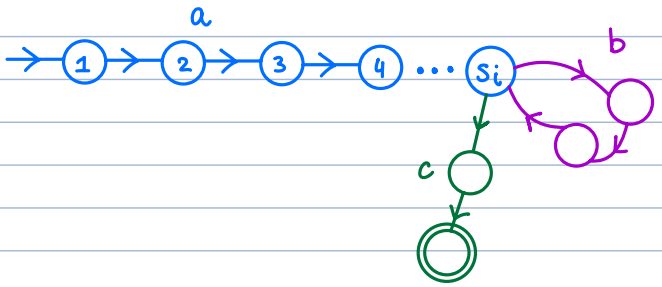
→ There are languages that are not computable by a DFAs <u>but</u> you cannot prove this by the pumping lemma.
→ "Myhill-Nerode Theorem" provides a full theorm to determine if a language is regular.

Proof of the Pumping Lemma:

- There exists a DFA D that computes L.
- let p = number of states in the DFA D.
- We need to see every x of length at least p can be split up as $x = a \cdot b \cdot c$ such that the previous properties hold. So lets take $x \geq p$:



By the pigeon hole principle we must have a repeat that is $S_i = S_j$ for some $i \neq j$ (Since we have more inputs than states, there will exist some input that must lead back to a previous state).
Therefore our DFA would look like:



From the picture we see:

$a \cdot c \in L$
$a \cdot b \cdot c \in L$
$a \cdot b \cdot b \cdot c \in L$
$\vdots$
$a(b)^i c \in L$

Formalizing the construction: let $S_0 = 0$ the starting state and let $S_i = T_D(S_{i-1}, x[i-1])$ $i = 1, \ldots, N$. As $N \geq P$, there must be two indicies i,j where $S_i = S_j$, and $i \neq j$, where $i \neq j \leq p$. Where:

$a = x[0]\, x[1] \cdots x[i-1]$ (could be empty)
$b = x[i]\, x[i+1] \cdots x[j-1]$ (cannot be empty)
$c = x[j] \cdots x[N-1]$ (could be empty)

Hence by the construction we form the conditions for our lemma:

→ (i) $a \cdot b^i \cdot c \in L$
→ (ii) length of b is not zero
→ length(ab) $\leq p$

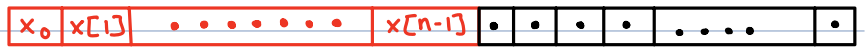
Moving on from DFA's and Boolean Circuits onto Turing Machines:

Issues with Boolean circuits: can compute everything with a <u>bounded</u> input.
Issues with DFAs: can compute on unbounded length... but we cannot compute everything.

Turing Machine ≡ DFA + left/right movement on input + Read/Write Memory

Anatomy of a Turing Machine:

→ Number of states is finite
→ There is an infinite memory tape
→ The head can write on the tape
→ The head can move left or right.



Turing Machines:

Defining each **computational step:**

→ You are in state i and read a bit α at the head of the tape.

→ Actions: → change state
→ write at position of the head
→ Move the head Left "L", Right "R", Stay "S"

Formalizing Turing Machines!

→ k States

→ $\sum \supseteq \{0, 1, \triangleright, \emptyset\}$ (finite alphabet)

→ "$\triangleright$" is notation for the start of the tape

→ "$\emptyset$" notation for nothing is written.

→ Transition function $\delta : \{0,1,\dots,k-1\} \times \sum \longrightarrow \{0,1,\dots k-1\} \times \sum \times \{L, R, S, H\}$

Right HALT
Left Stay

δ(State i, a) = (State j, b, "Movement of head")

symbol from alphabet

symbol from alphabet

Computation:

Start with head at x[0]
State "0" (Starting state) = current_state
Repeat:
(new_state, new_symbol, A) = δ(current_state, Tape[Head])
Update: current_state = new_state
Tape[Head] = new_symbol
Action:
If A=L; Head = max(0, Head-1)
If A=R; Head = Head+1
If A=S; Head = Head
If A=H; Then stop.

On an input x:

→ if M halts on input x:
M(x) = Tape[0]...Tape[Head]
Memory contents till local of head

→ if M does not halt: M(x) = "$\perp$"

A function $f: \{0,1\}^* \longrightarrow \{0,1\}^*$ is computed by a Turing Machine M if $f(x) = M(x)$ for all x.

A Language L is recognized by a TM if $M(x) = 1$ where $x \in L$, $M(x) = 0$ where $x \notin L$ and it HALTS on x.

Example Turing Machine:

$k = 1$; $\sum = \{0, 1, \triangleright, \emptyset\}$ $\Rightarrow$

$S_M(0,0) = (0, 1, R)$
$S_M(0, 1) = (0, 0, R)$
$S_M(0, \emptyset) = (0, \emptyset, H)$

Flips the input

Remarks about Turing Machines:

TMs are esserntially a programming language. We can also implement random acess of an index in an array using a turing machine.

For every program P, $\exists$ a TM such that $P(x) = M(x)$. If P takes T time → M takes T^2 time

A function $f: \{0,1\}^* \to \{0,1\}^*$ is COMPUTABLE if there is a TM M where $M(x) = f(x)$ for all x

Writing the MAJ using a Turing Machine:

→ Main idea: Trying to match 0's and 1's. If we cannot match a zero then there are more zero's than one's.

Steps:

① "Scan" to the right until you find a 0. here our alphabet is $\sum = \{0, 1, a, \triangleright, \emptyset\}$ ↳ "Seen"

② If No "0" is found
- → Clean up the tape and return 1.

③ If a "0" is found:
- → Mark the "0" as seen
- → Go to start of the input
- → Scan the input to find a "1"
- → If 1 found:
  - Mark this 1 as seen
  - Go to start
- → If 1 not found:
  - Clean up and return 0.

Writing Palindrome: $\{0,1\}^* \longrightarrow \{0,1\}$ using a Turing Machine

Idea: "Check bits at both ends, if they match mark them, and look for next"

→ We can remember what the first bit is using the state
"Mark first bit as scene"

→ Keep scanning right until end:
- → Move one step to left
- → Try to match; if yes
  - → go to first unmarked symbol
  - → Repeat
- → If NO:
  - Cleanup tape, output 0, HALT

$\delta(0,0) = (\text{GoEnd0}, a, R)$
$\delta(0,1) = (\text{GoEnd1}, a, R)$
] Remember symbol mark as read

$\delta(\text{GoEnd0}, 0) = (\text{GoEnd0}, 0, R)$ → keep moving
$\delta(\text{GoEnd0}, 1) = (\text{GoEnd0}, 1, R)$ → to find end

$\delta(\text{GoEnd0}, \emptyset) = (\text{check0}, \emptyset, L)$ → found end
$\delta(\text{check0}, 0) = (\text{GoStart}, \emptyset, L)$ → found match
$\delta(\text{check0}, 1) = (\text{Write0}, 1, L)$ → Clean up

$\delta(\text{GoStart}, 0) = (\text{GoStart}, 0, L)$
$\delta(\text{GoStart}, 1) = (\text{GoStart}, 1, L)$
$\delta(\text{GoStart}, a) = (\text{"0"}, a, R)$
] Find first unmarked position

On the right we see that our transition has a certain modularity
Note we would have to still implement goEnd1, Write0, and Accept.

Going Beyond Turing Machines??

- We could add multiple heads for the machine
- Adding multiple tapes for a machine.
- ALL modifications can be simulated by our plain Turing Machine

Church-Turing Thesis: Every function that is computable by physical means is computable by a TM!

Have our Cake and eat it too Principle: (HOCAEIT Principle)

Having the Cake: to show something is computable we use a high level programming language
Eating it: we need to show a turing machine cannot compute something then it is uncomputable.

We have the ability to both prove something using something very powerfull and to disprove something we simply need to use our weakest form of computation.

Universality:

- There is a single universal turing machine that can simulate all turing machines ≡ "Universal TM"
- Universal TM ≅ "Compiler + Executor" ⇒ $U_{TM}(M,x) = M(x)$ (x → input; M → Machine (represented as binary strings))
- Recall a TM = $\delta[k]\times\sum \longrightarrow [k]\times\sum\times\{L,R,S,H\} \longrightarrow \delta(i,a) = (j,b,\{L,R,S,H\})$
- So how can we encode δ as $\{0,1\}^*$ ⟶ we just use a PFE encoding similar to the ones used when encoding boolean circuit.

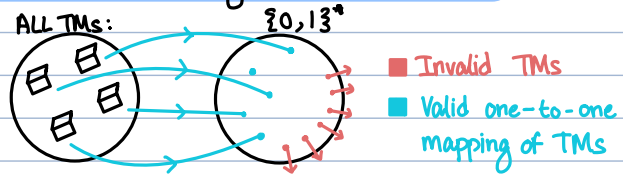
⟶ $(k, \ell, (0,a_0,-,-,-),(0,a_1,-,-,-),\dots,(0,a_{\ell-1},-,-,-) \Rightarrow$ There will be $k\cdot\ell$ tuples.
$(1,a_0,-,-,-),(1,a_1,-,-,-),\dots,(1,a_{\ell-1},-,-,-)$ ⟶ Each tuple has 5 integers, and each tuple can be a encoded as a $\{0,1\}^*$
$\vdots$
$(k,a_0,-,-,-),(k,a_1,-,-,-),\dots,(k,a_{\ell-1},-,-,-))$

⟶ length of encoding is: $k\cdot\ell$ tuples ⇒ each tuple has 5 ints ⇒ $O(\log k + \log \ell)$ bits ⇒ each integer is $\max(k,\ell)$

⟶ The length of the encoding is $O(k\ell(\log(k)+\log(\ell))$

⟶ A TM M can be represented as a binary string. ⇒ let <M> denote the binary representation of M

Notational Convention: if a string $\alpha \in \{0,1\}^*$ is not a valid turing machine, just set it to a trivial turing machine that always outputs 0.



Eval: $\{0,1\}^* \longrightarrow \{0,1\}^* \cup \perp$; Turing states eval is computable. There exists a TM U such that $U(M,x) = Eval(M,x) = M(x)$

Using the cakee principle to prove universality: ⟶ we can just implement it in python!

```
def EVAL(δ,x):
    Tape=[" "] + [a for a in x]
    i=0; s=0  # i= head pos, s=state
    while True:
        S, Tape[i], d = δ[S, Tape[i]]
        if d== "H": break
        if d== "L": i= max(i-1,0)
        if d== "R": i+= 1
        if i>= len(Tape): Tape.append('∅')

    j=0; y=[] # output
    while Tape[j] != '∅'
        y.append[Tape[j]; j+=1;
```

⟶ Since we wrote a python program for EVAL hence there is a turing machine for EVAL which means there is universal turing machine.

⟶ There exists universal TMs with 25 states and alphabet $\{0,1,\emptyset,\triangleright\}$.

⟶ Meta-circular evaluator (interpreter/compiler) GCC is in C

⟶ Universality transcends the specific model: ∃ a python program that computes and executes all Java Programs

⟶ "Turing-Complete Languages": All programming languages that can simulate a universal TM. ex: Latex, HTML+CSS, ...,

Uncomputability:

Thm: There is a function $f: \{0,1\}^* \to \{0,1\}^*$ that cannot be computed by a TM.

Toddler function:

Todd : $\{0,1\}^* \to \{0,1\}$ ⟹ Toddler does the opposite of what you ask it to do

$$\text{Todd}(\langle M \rangle) = \begin{cases} 1 & \text{If } M(\langle M \rangle) \text{ halts and the first bit of of output is zero} \\ 0 & \text{If } M(\langle M \rangle) \text{ halts and the first bit of output is one} \\ 0 & \text{If } M(\langle M \rangle) \text{ does not halt} \end{cases}$$

Thm: Toddler is not computable

Proof by Contradiction: suppose we have a program and turing machine that computes toddler called N. ⟹ $N(\langle N \rangle) = \text{Todd}(\langle N \rangle)$

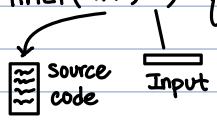
Case #1: What should N do when given $\langle N \rangle$ as input?
If $N(\langle N \rangle)$ halts and output is 1
then $\text{Todd}(\langle N \rangle) = 0$ by definition

Case #2: If $N(\langle N \rangle)$ is 0
Then $\text{Todd}(\langle N \rangle) = 1$.

This results in a contradiction therefore toddler is not computable.

HALTING PROBLEM:

$\text{HALT}: \{0,1\}^* \to \{0,1\} \Rightarrow \text{HALT}(\langle M \rangle, x) = \begin{cases} 1 \text{ if } M \text{ halts when run on } x \\ 0 \text{ else} \end{cases}$; HALT is also uncomputable



Proof: if HALT was computable, then TODD would be computable. Since TODD is not computable then halt is not computable.

suppose we have a black box that computes HALT.

→ we then use this to solve toddler

Solving TODD using HALT ⇒ input $\langle M \rangle$: Output: $\text{TODD}(\langle M \rangle)$

if $\text{HALT}(\langle M \rangle, \langle M \rangle) = 1$, then
→ Run M on $\langle M \rangle$
→ output opposite of first bit
if $\text{Halt}(\langle M \rangle, \langle M \rangle) = 0$, then
→ output 0.

⟶ So the above algorithm would compute Toddler if we have a program for HALT!
Therefore HALT itself is uncomputable.

Goldbach Conjecture:

→ Every even number can be written as a sum of two primes.

```
def(isPrime(n)):
    for a in range(2,n):
        if n % a == 0:
            return 0
    return 1.
```

```
def Goldbach(S):
    n=4.
    while(true):
        GoldbachN = False
        For p in range(2,n):
            if isPrime(p) ∧ isPrime(n-p)
                GoldbachN=true
                break
        if GoldbachN == True:
            n=n+2
        else:
            return false
```

if our goldbach function halts then we would know the whether or not the goldbach conjecture is true or not.

Even for such few lines we see that halt is a very hard problem to solve.

Turing Cook Reductions:

Reduce from problem A to problem B:

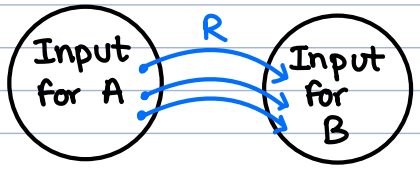
→ If I can solve B, then I can solve problem A
→ If B is computable then A is computable
→ If A is uncomputable then B is uncomputable

"We can use the black box of B multiple times."

Thm: If $f: \{0,1\}^* \to \{0,1\}$ is uncomputable, then $NOT(f): \{0,1\}^* \to \{0,1\}$ is uncomputable. → We can reduce "f" to "NOT(f)"

KARP Reduction:

Reducing Problem A to problem B; KARP reductions work by transforming input.



aka: $R: \{\text{inputs for A}\} \to \{\text{inputs for B}\}$
$R: \{0,1\}^* \to \{0,1\}^*$ → Specification (WHAT)
$\forall x$ where $A(x) = B(R(x))$ → Utility (WHY)
Need to show R is computable → Implementation (HOW)

Suppose we have a reduction function R as above
→ if B is computable, then A is computable
→ if A is uncomputable, then B is uncomputable

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

$$\text{Haltonzero}(\langle M \rangle) = \begin{cases} 1 \text{ if M halts on input 0} \\ 0 \text{ if M does not halt on input 0.} \end{cases}$$

Hence HALTONZERO is also uncomputable

R((M,x)) = def N(z):
    Run EVAL(M,x)

HALT(M,x) = HALTONZERO(R(M,x))
→ if M halts on x, then N halts on *all* inputs so it would halt on zero
→ if M does not halt on x, then N will output zero.

An Alternate Proof for HALT:

Suppose we have a machine H that computes Halt:

```
def cantSolveMe(<M>):
    if H(<M>, <M>) == 1:
        while true:
            a=1
    else: #H(<M>,<M>)==0
        return 0
```

- Suppose we give CANTSOLVEME itself: CANTSOLVEME(<CANTSOLVEME>)
- If it halts and gives an output → then HALT returns 1 and we would enter the while loop
- If it does not halt → HALT then returns zero and we enter else → and we do output and halt.

Another Reduction:

$$\text{NotEmpty}(\langle M\rangle) = \begin{cases} 1 \text{ if there is some } x \text{ s.t. } M(x)=1 \\ 0 \text{ else} \end{cases}$$

Can we prove that notempty is uncomputable

R =

```
def N(z):
    Run EVAL(<M>, 0)
    returns 1
```

Suppose M halts on input zero then z returns 1 on all inputs. Hence NOTEMPTY(<N>) = 1.

Suppose M does not halt on input zero then N(z) will also not halt and return any value. Hence N does not have any string producing 1. Which means NOTEMPTY(<N>) = 0.

→ Since HALTONZERO(<M>) = NOTEMPTY(R(<M>)) which means that not empty is uncomputable.

Software Verification:

Given a program A, B can we determine if they are equivilent...?

→ SEMANTIC PROPERTY:
- → Definied as where the property only depends on input/output behaviour or functionality of the program/TM.

→ Semantic Function:

$F: \{0,1\}^* \to \{0,1\}$ (programs of TMs)

we say M, M' are equivilent if $\forall x \quad M(x) = M(x')$

then F is semantic if for all equivilent programs M' and M then $F(M) = F(M')$

- → HaltOnZero, and isMajority are semantic
- → TODD is not semantic
  even if M, M' are equivilent, TODD runs on their source code and does not look at the input/output.

Rice's Theorm:

Any non-trivial symantic function cannot be computed ⟶ true software verification cannot be achieved.

Looking at an example of a semantic function:

isMajority we use a reduction from halt on zero:

```
def N(z):
    Run Eval(M,0)
    Return MAJORITY(z)
```

→ If HALTONZERO(<M>) = 1:
M halts on zero then N computes majority.

→ If HALTONZERO(<M>) = 0:
M does not halt on zero then N is not computing majority.

Proof of Rice's theorm:

Supposse $F: \{0,1\}^* \rightarrow$ is a non-trivial semantic property. ⟶ ∃ some machine M_0 such that $F(M_0)=0$
∃ some machine M_1 such that $F(M_1)=1$

→ let INF be a program that goes into an infinite loop

Case 1: $F(INF)=0$.
We can show that HALTONZERO(M) = F(R(M))
→ If M does not halt on zero: R(M) ≡ INF
→ If M halts on zero: $R(M)=M_1$
→ If HALTONZERO(M) = 1 then $F(R(M))=F(M_1)=1$.
→ If HALTONZERO(M) = 0 then R(M) is equivilent to INF. $F(R(M))=F(INF)=0$.

R(<M>) ≡
```
def N(x)
    → Run Eval(M,0)
    Return Eval(M,x)
```

Case 2: $F(INF)=1$.
→ We can do a reduction using NOTHALTONZERO(M) = F(R(M))

Hence by using reductions we have shown that Rice's theorm is true.

Turing Completeness

→ All programming languages that can simulate a universal Turing Machines:
→ A feature (can use it to program anaything)
→ A Bug (Reasoning about programs becomes unsolvable)

→ "Restricted Computational Models": weaker than TMs but we can reason about other meta questions.
→ An example of restricted computational models: Context-Free-Grammers (CFGs)

Context Free Grammers (CFGs)

DFAs — CFGs — TMs

Power ⟶

A grammar $G = (V, \Sigma_1, R, S)$ ⇒ Σ is disjoint from variables.

- Σ_1: Alphabet/Terminals
- V: Variables
- R: Rules
- S: Start variables

Ex: G_1: $(\{A\}, \{0,1\}, R, A)$

Rules: $A \to 0A1$ hence we see that: $A \to 0A1$ or ε

$A \to \varepsilon$

---

we have **derivations** suppose $\alpha \in (\Sigma \cup V)^*$ and $\beta \in (\Sigma \cup V)^*$. We say $\alpha \Rightarrow_G \beta$ if we can get β by replacing a variable in α using a rule from R

we can **chain derivations** we say $\alpha \Rightarrow^*_G \beta$ is there exists a chain of derivations from alpha to β: ($\alpha, \beta \in (\Sigma \cup V)^*$) in-other words we see $\alpha \Rightarrow_G \alpha_1 \Rightarrow_G \alpha_2 \Rightarrow_G \alpha_3 \Rightarrow \dots \Rightarrow_G \beta$

ex using G_1: $A \Rightarrow^*_G 00A11$

$A \Rightarrow 0A1 \Rightarrow 00A11$

$AA \Rightarrow^*_G 0001110A1$

$AA \to 0A1A \Rightarrow 00A11A \Rightarrow 000A111A \Rightarrow 000111A \Rightarrow 0001110A1$

If G is a grammar, $x \in \Sigma^*$ is **matched** to G_1 is the start symbol $S \Rightarrow^*_G x$

$L_G = \{ x \in \Sigma^* : x \text{ matches } G \}$

$F_G : \Sigma^* \to \{0,1\}$

Context Free Grammars: a function f: is context free if $\exists$ a grammar G such that $f(x) = F_G(x)$ $\forall$ x.

Continuing with G_1 example:

$L_G = \{0^n 1^n : n \geq 0\}$ since our start can only be A the AA example doesn't work

Recall that $\{0^n 1^n\}$ was not regular but it is context free.

Take G_2 to be:

G_2: $(\{A\}, \{0,1\}, R, A)$

$A \to 0A0$ and $A \to 0, 1, \varepsilon$

$A \to 1A1$

the grammar G_2 generates: $L_G \{x : x \text{ is a palindrome}\}$

if we have multiple rules for same variable we write it as $A \to z_1 | z_2 | z_3$

A more Complex Grammar:

G_3: $(\{S\}, \{"(", ")"\}, R, S\}$ ⇒ $L_{G_3} = \{$ All strings of "well matched" paranthesis expressions.

$S \to (S) | S\_S | \varepsilon$

Grammar of Programming Languages:

⟶ $\exists$ a grammar G that generates "syntactically" valid program for each language: C++, Python, HTML, Java

Regular Expressions and Grammars:

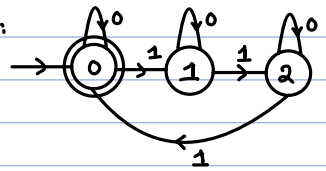
Every regular language is also context-free. For every DFA D, ∃a CFG, G such that $D(x) = F_G(x)\ \forall x$.

Proof:

DFA ≡ $(T, \vec{S})$ ← accepting states

↳ transition $t: [C] \times \{0,1\} \to [C]$

Ex:



Create a variable for each state: $V = \{V_0, V_1, V_2, \dots, V_{C-1}\}$ are the variables

- $\sum = \{0,1\}$, and starting variable is V_0

Rules: for each transition $T(i,a) = j$ and add a rule $V_i \to aV_j$, to deal with an accepting states i in the DFA, add $V_i \to \varepsilon$

Taking the ex above:

$\to V = \{V_0, V_1, V_2\}$ Rules: $V_0 \to 0V_0 \mid 1V_1 \mid \varepsilon$

$V_1 \to 0V_1 \mid 1V_1$

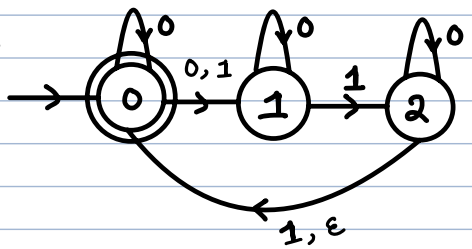
$V_2 \to 0V_2 \mid 1V_0$

CFGs are strictly more powerfull than DFAs

Converting from NFAs to CFGs:

$\to V: \{V_0, V_1, V_2\}$ starting variable is V_0

Ex:



$V_0 \to 0V_0 \mid 0V_1 \mid 1V_1 \mid \varepsilon$ (for multiple 0,1 transitions we can just add that as an option)

$V_1 \to 0V_1 \mid 1V_1$

$V_2 \to 0V_2 \mid 1V_0 \mid V_0$ (for ε we write a rule to the next variable without any alphabet)

Tips for Designing CFGs:

→ Breakup the language L into simple pieces:

$G_1: L_1$, $G_2: L_2$ } → $L = L_1 \cup L_2$ let $S_1 \equiv$ start symbol for G_1 and $S_2 \equiv$ start symbol for G_2 → add rule: $S \to S_1 \mid S_2$

→ CFGs can "link strings" ex: $A \to 0A1$

→ Regular languages are context-free

Example: $ADD = \{0^m 1 0^n 1 0^{n+m} \mid m,n \geq 0\}$

Idea: pair up the outer zeros, then pair inner zeros.

0^n	1	0^m	1	0^m	0^n

$A \to 0A0$ we see $A \Rightarrow^* 0^m A^m \Rightarrow 0^m 1 B 0^m \to 0^m 1 0^n 1 0^n 0^m$

$A \to 1B$

$B \to 0B0$ $V = \{A, B\}$, $\sum = \{0,1\}$, start = A

$B \to 1$ Rules: $A \to 0A0 \mid 1B$ & $B \to 0B0 \mid 1$

Ex #2: $L = \{x : x \text{ is of even length \& middle two symbols are same}\}$

$A \to BAB \mid 11 \mid 00$

$B \to 0 \mid 1$

Ex #3: $L = \{x : \text{odd length, first, middle, end are same}\}$

$L_1 = \{\text{odd length with first} = 0\}$

	L_1
$\Rightarrow A \to 0B0 \mid 0$	$D \to 1E1 \mid 1$
$B \to cBc \mid 0$	$E \to gEg \mid 1$
$C \to 0 \mid 1$	$g \to 0 \mid 1$

CHOMSKY NORMAL FORM (CNF)

This form can be very usefull to determine if a string x is part of the grammar G.

A grammar G is in CNF:

① Every Rule is of the form:

$A \to BC$ (A, B, C are variables)

$A \to a$ (a is a terminal)

One exception: we can have a rule $S \to \varepsilon$.

② Start Symbol doesn't occurr on the right hand side

Examples of rules that are not allowed:

$A \to ABa$ $\quad A \to \varepsilon$

$A \to 00$ $\quad A \to B$

$A \to Bo$

$A \to oB$

Thm 1: For every grammar G, there exist another grammar G_{CNF} such that it is in CNF form and $\forall x \; F_G(x) = F_{G_{CNF}}(x)$.

Thm 2: If G is in CNF, then we can compute $F_G(x)$.

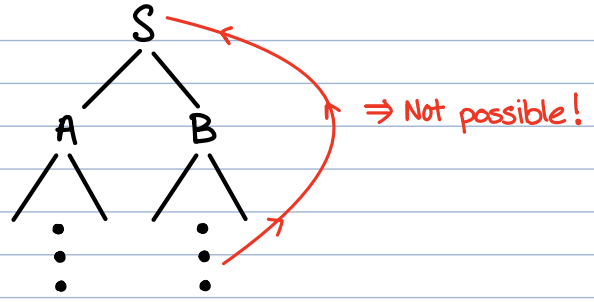
Proof of theorm #2:

$G = (V, \Sigma, R, S)$ and $x \in \Sigma^*$

```
def Check(A, y):

  if len(y) == 1:
      If A→y is a rule
          Return 1
      else:
          Return 0
  if len(y) ≥ 2:
      for every rule A→BC:
          for every split of y = y1∪y2:

              if Check(B, y1) == 1 ∧ check(C, y2) == 1
                  Return 1
  Return 0
```



Having the CNF allows us to ensure when we use our recursion we will always make progress since we cannot return to the start state...

Semantic Properties of CFGs:

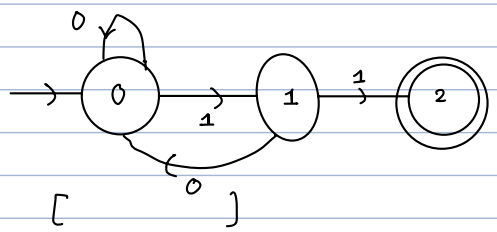
$F: \{0,1\}^* \longrightarrow \{0,1\}$ takes a grammar to see if it is possible:

There exists semantic functions that are computable.

NOTEMPTY: 1 if $\exists x \in L_G$ it is computable

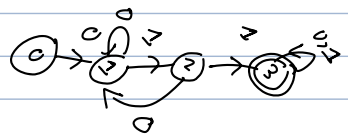
Howevers: ISFULL= 1 if $L_G = \Sigma^*$ is uncomputable

11



[ ]

```
for (int i=0, i<size; i++)
    if [arr[i] != 1)
        if
```



Pumping Lemma for CFGs:

For every grammar G, $\exists$ a number p such that $\forall x \in L$, $|x| \geq p$, x can be written as: $x = \boxed{a}\boxed{b}\boxed{c}\boxed{d}\boxed{e}$ such that:

1. $ab^icd^ie \in L$ for all $i \geq 0$
2. $|bd| > 0$
3. $|bcd| \leq p$

Ex: $L = \{0^n1^n2^n : n \geq 0\}$ is not context free.

take $x = 0^p1^p2^p$ since the length of bcd can contain atmost p it will only have $\{0,1\}$ or $\{1,2\}$ therefore in either case pumping will cause there to be more of 1's than either the # of 0s or the # of 2's

Ex: Double = $\{w;w \mid w \in \{0,1\}^*\} \Rightarrow x = 0^p1^p;0^p1^p$

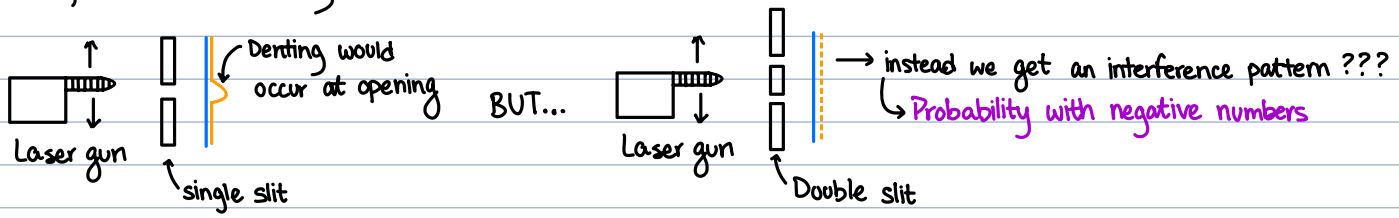
case bcd on left hand side of ; there will be unequal amount of values on the left/right side

case bcd in middle: unequall #1s and zero's

case semicolon: multiple semi-colons.

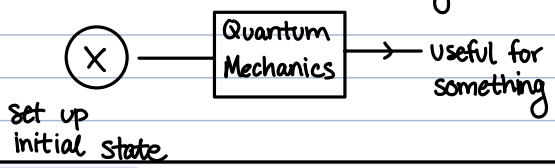
Quantum Weirdness:

Imagine the following scenarios:



Nature is quantum. There is no obvious algorithm to simulate a large quantum system efficiently (exponential time)

Idea: if we cannot simulate a system, then the system can be seen as performing powerful computations.



Q-BIT:

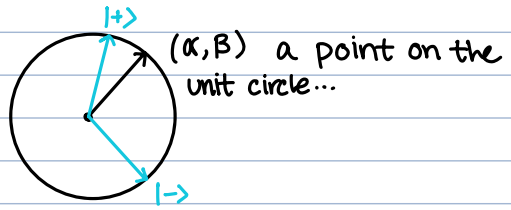
0 → low Voltage
1 → High Voltage

Electrons "Spin"	"Photons"
"Up" is 0, "Down" 1	Horizontal ↔ ; Vertical ↕

Quantum Mechanic Law 1: If a system can be in one of two states $\langle 0\rangle, \langle 1\rangle$, then it can also be in "ket" which is a superposition of the two: $\alpha|0\rangle + \beta|1\rangle$ where $\alpha^2+\beta^2=1$.

ex: $0.8|0\rangle + 0.6|1\rangle$
$0.8|0\rangle - 0.6|1\rangle$
Different! → this is a q-bit

$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \equiv |+\rangle$
$\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle \equiv |-\rangle$



What can we do with a Q-Bit

Measuring a Q-Bit: photon $\sim |\psi\rangle$ → "HORIZ" → Horizontal polirization is α^2; Vertical polirization is β^2

↳ $|0\rangle \equiv$ Horizontal ; $|1\rangle \equiv$ vertical polarization

Quantum Mechanic Law 2: The measurment device says "H" with probability α^2 and "V" with probability β^2. There is no way to know α or β without changing them → uncertainty principle. (Only outputs H or V)

Quantum Mechanic Law 3a: We can build a device that rotates the state by an angle θ

$|0\rangle \sim \boxed{R_\theta} \to \cos\theta|0\rangle + \sin\theta|1\rangle$; $|1\rangle \sim \boxed{R_\theta} \to -\sin\theta|0\rangle + \cos\theta|1\rangle$

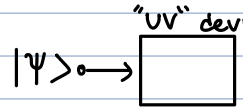
Quantum Mechanic Law 3b: We can build a device that reflects the state about any line through the origin.

Ex: Reflection around the line with angle $22.5^\circ \equiv$ Hadamard Gate.

$|0\rangle \sim \boxed{\text{Hadmard gate}} \to \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ $\quad |1\rangle \sim \boxed{\text{Hadmard gate}} \to \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$

A more complete QM Law #3:

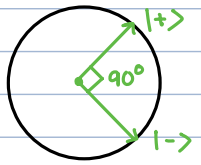
For any two perpendicular vectors $|u\rangle, |v\rangle$ on the unit circle:



we get $|u\rangle$ with probability (length of projection onto $|u\rangle$)^2
we get $|v\rangle$ with probability (length of projection onto $|v\rangle$)^2

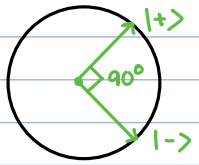
Ex: $|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ → let this be u,v

$|-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$



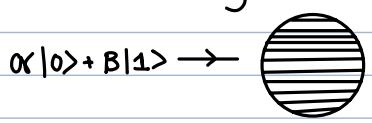
We get $|+\rangle$ with prob ½ and $|-\rangle$ with prob ½

Ex#2:

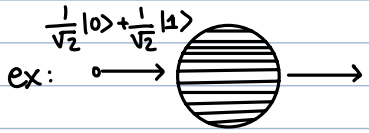


We get $|+\rangle$ with prob ½ and $|-\rangle$ with prob ½

Horizontal Polarizing Filter:

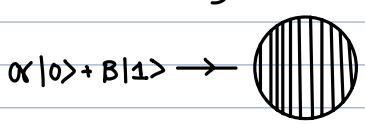


1. Measures in $|0\rangle$ & $|1\rangle$ basis
2. If outcome $|0\rangle$ lets it through
3. If outcome $|1\rangle$ photon converted to heat.

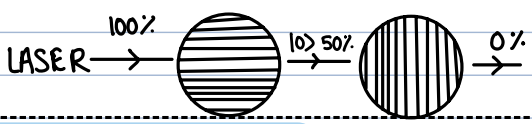


$|0\rangle$ 50% of the time
$|1\rangle$ to heat 50% of the time

Vertical Polarizing Filter:

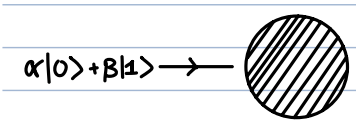


produes $|1\rangle$ with β^2 probability
heat α^2 probability.

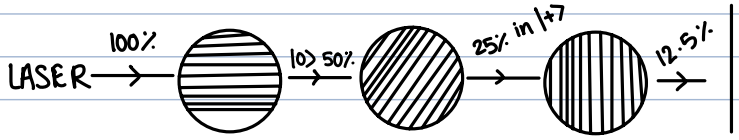


No light is produced

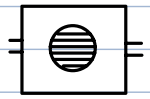
Diagnal Polarizing Filter:



Measures in $|+\rangle$ or $|-\rangle$ basis
If outcome is $|+\rangle$ it passess
If outcome is $|-\rangle$ converts to heat

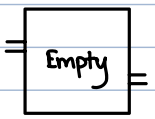


EV Bomb:



OR

if we generate heat we blow up



Suppose we send a photon of $|+\rangle$
→ if it's empty we get $|+\rangle$
→ If it's bomb → $|0\rangle$ with $1/2$ or $|1\rangle$ with $1/2$ it blows up

---

Idea: Horizontal Filter

Case empty: with $|+\rangle \longrightarrow |+\rangle \Rightarrow \{|0\rangle$ with $1/2$; $|1\rangle$ with $1/2\}$

Case Bomb: $|+\rangle \longrightarrow$ 50% chance we blow up
$\longrightarrow$ 50% chance : $\{|0\rangle$ with prob 1 and $|1\rangle$ with $0\}$

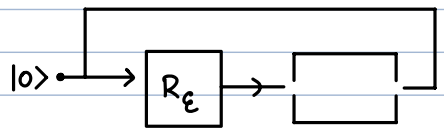
Idea: Measurment "H" device

Case Dud: final readout is $|+\rangle$ with prob 100%

Case Bomb: $\longrightarrow$ 50% blow up
$\longrightarrow$ 50% with $|+\rangle$ with prob $1/2$ & $|-\rangle$ with prob $1/2$.
↳ 25% inconclusive, 25% detection

A better solution:

We pick a number $n=1000$ where $\epsilon = \frac{2\pi}{n}$ and we build a device the Rotates he ϵ : R_ϵ

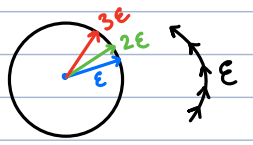


Rotate by ϵ
Send through box } Repeat N times
Measure in "Horiz" device

Case Dud:

#1 iteration : In: $\cos(\epsilon)|0\rangle + \sin(\epsilon)|1\rangle$
#2 iteration: changes by ϵ
#n^{th} iteration: $|1\rangle$

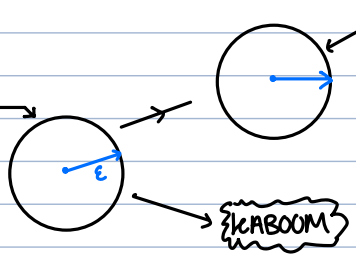
Measuring we get $|1\rangle$ with 100% and $|0\rangle$ with 0%



rotates by ϵ each iteration.

Case Bumb:

#1 iteration: ⟶
#2 iteration repeat:
explosion prob $\leq \left(\frac{\pi}{2n}\right)^2$
we ever explode $\leq n \cdot \left(\frac{\pi}{2n}\right)^2$



Reset to $|0\rangle$ after "bomb" with prob $(\cos\epsilon)^2$

with prob $(\sin\epsilon)^2$

If we don't explode after n iteration we are in state $|0\rangle$. ⇒ We explode with 0.00247 prob
With 0.99247 prob we detect bomb

Final Conclusion:

→ With quantum computing we have the ability to compose operations which is very powerful

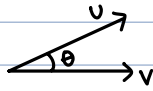
A Quick Math Review:

$\vec{V} = \begin{bmatrix} -2 \\ 1 \\ 3 \end{bmatrix} \Rightarrow -2\vec{i} + \vec{j} + 3\vec{k} \longrightarrow -2\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + 3\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \longrightarrow V = -2e_1 + e_2 + 3e_3$

$\vec{V} \Rightarrow$ in quantum notation: $V = \alpha|1\rangle + \alpha|2\rangle + \alpha|3\rangle + \ldots + \alpha_d|d\rangle$ → "KET" Dirac notation

$U \cdot V \equiv$ dot product: $U \cdot V = U_1V_1 + U_2V_2 + \ldots + U_dV_d \equiv \langle U|V\rangle$ ← KET (column vector)

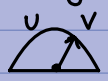
Magnitude: $\|U\| = U \cdot U \Rightarrow U_1^2 + U_2^2 + \ldots + U_d^2$ ← Bra (Row vector)

Geometric view: 

then $U \cdot V = \|U\| \cdot \|V\| \cdot \cos\theta$

If U and V are unit vectors and θ is 90° then U & V are orthonormal

Then: for any orthonormal vectors $|U\rangle$ and $|V\rangle$ are orthonormal $\in \mathbb{R}^2$ can build:



⟶ $|U\rangle$ with probability $\langle U|\psi\rangle^2$
$|V\rangle$ with probability $\langle V|\psi\rangle^2$

Matrix Multiplication:

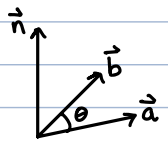
$A = [a_{ij}]_{m\times n}$ and $B = [b_{ij}]_{n\times p}$ then their product $AB = C = [c_{ij}]_{m\times p}$

where $c_{ij} = \sum_{r=1}^{n} a_{ir}b_{rj} \Rightarrow \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{bmatrix} = \begin{bmatrix} 1\cdot5+2\cdot8 & 1\cdot6+2\cdot9 & 1\cdot7+2\cdot10 \\ 3\cdot5+4\cdot8 & 3\cdot6+4\cdot9 & 3\cdot7+4\cdot10 \end{bmatrix} \Rightarrow \begin{bmatrix} 21 & 24 & 27 \\ 47 & 54 & 61 \end{bmatrix}$

Transpose:

$\begin{bmatrix} 0 & 4 \\ 7 & 0 \\ 3 & 1 \end{bmatrix}^T \longrightarrow \begin{bmatrix} 0 & 7 & 3 \\ 4 & 0 & 1 \end{bmatrix}$ and $(A^T)^T = A$

Cross Product:



$\vec{a} \times \vec{b} = |a||b|\sin(\theta)[\vec{n}]$ observe that: $\vec{a} = a_1\vec{i} + a_2\vec{j} + a_3\vec{k}$; $b = b_1\vec{i} + b_2\vec{j} + b_3\vec{k}$

then $\vec{c} = \vec{a} \times \vec{b} = \vec{i}|a_2b_3 - a_3b_2| - \vec{j}|a_1b_3 - a_3b_1| + \vec{k}|a_1b_2 - a_2b_1|$

Multiple Qubits:

Qutrit: A system with three possible options $|1\rangle, |2\rangle, |3\rangle$
Qudit: A system with "d" possible options $|1\rangle \dots |d\rangle$

Photon ... ⊗ with $\{\alpha_1, \beta_1\}$ and ⊗ with $\{\alpha_2, \beta_2\}$
New QM Law 1: if a system has d options then it can be a superposition of all of them
$|\psi\rangle = \alpha_1|1\rangle \dots \alpha_d|d\rangle$ where $\alpha_1^2 + \dots + \alpha_d^2 = 1$

If we have 2 photons then we have four "kets" $|00\rangle, |01\rangle, |10\rangle, |11\rangle$

New QM Law 2: we measure using a standard basis

$|\psi\rangle \longrightarrow$ "meas" $\longrightarrow \{|1\rangle$ with prob α_1^2, $|2\rangle$ with prob $\alpha_2^2 \dots |d\rangle$ with prob $\alpha_d^2\}$

New QM Law 3: For any linear transformation $U: \mathbb{R}^d \to \mathbb{R}^d$ that preserves lengths i.e. $\forall\ \vec{v}$ if $\|\vec{v}\| = \|U(\vec{v})\|$ we can build a device that simulates U.

Gates:

Any linear transformation that preserves length (QM law 3)

$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \Rightarrow \left[\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right]$

Ex #1: "Not" Gate

$\text{NOT}(|0\rangle) = |1\rangle$
$\text{NOT}(|1\rangle) = |0\rangle$
$\text{NOT}(\alpha\cdot|0\rangle + \beta|1\rangle) = \beta|0\rangle + \alpha|1\rangle$

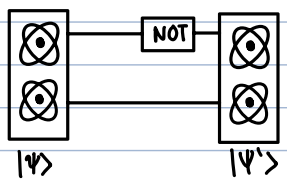
$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} \alpha \\ \beta \end{bmatrix} = [\beta, \alpha]$

Ex#2: Hadamard Gate:

Reflection about line with 22.5° $\longrightarrow \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}\begin{bmatrix} \alpha \\ 1 \end{bmatrix}$

$|0\rangle \to |+\rangle$
$|1\rangle \to |-\rangle \Rightarrow \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$
$\alpha|0\rangle + \beta|1\rangle \to \alpha\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{2}|1\rangle\right) + \beta\left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{2}|1\rangle\right)$
$\longrightarrow \frac{\alpha+\beta}{\sqrt{2}}|0\rangle + \frac{\alpha-\beta}{\sqrt{2}}|1\rangle$

Multiple Qubits:

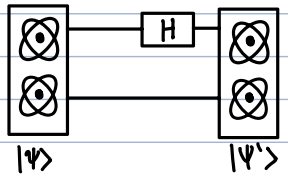


with α_{00} → {Photon 1 is in state $|1\rangle$, Photon 2 is in state $|0\rangle$}
with α_{10} → {Photon 1 is in state $|0\rangle$, Photon 2 is in state $|0\rangle$}
with α_{01} → {Photon 1 is in state $|1\rangle$, Photon 2 is in state $|1\rangle$}
with α_{11} → {Photon 1 is in state $|0\rangle$, Photon 2 is in state $|1\rangle$}

$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

$[a_1, b_1, a_2, b_2] \Rightarrow \{|1\rangle, |0\rangle, |1\rangle, |0\rangle\}$

$4\times4 \cdot 4\times1 \Rightarrow 4\times1$



$|00\rangle \to \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$
$|10\rangle \to \frac{1}{\sqrt{2}}|00\rangle - \frac{1}{\sqrt{2}}|10\rangle$
$|01\rangle \to \frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|11\rangle$
$|11\rangle \to \frac{1}{\sqrt{2}}|01\rangle - \frac{1}{\sqrt{2}}|11\rangle$

$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ 1/\sqrt{2} & -1/\sqrt{2} & 0 & 0 \\ 0 & 0 & 1/\sqrt{2} & 1/\sqrt{2} \\ 0 & 0 & 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ 1 \\ 0 \end{bmatrix}$

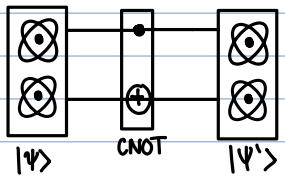
$\begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$

CNOT operation:

Classical: CNOT(a,b) if a=0 then b goes true $\longrightarrow a \oplus b$

If a=1 then b flipps

Quantum:

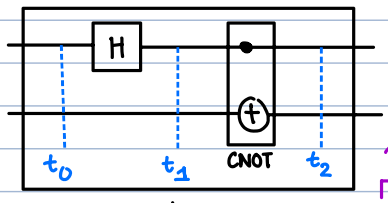


$|00\rangle \longrightarrow |00\rangle$
$|10\rangle \longrightarrow |11\rangle$
$|01\rangle \longrightarrow |01\rangle$
$|11\rangle \longrightarrow |10\rangle$

$\longrightarrow \alpha_{00}|00\rangle + \alpha_{10}|10\rangle + \alpha_{01}|01\rangle + \alpha_{11}|11\rangle \longrightarrow \alpha_{00}|00\rangle + \alpha_{10}|11\rangle + \alpha_{01}|01\rangle + \alpha_{11}|10\rangle$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Quantum Circuits:



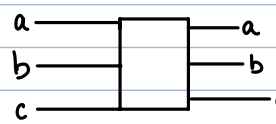
Matrix Notation: $|\psi^2\rangle = CNOT \cdot |\psi^1\rangle = CNOT \cdot H \cdot |\psi^0\rangle$

"EPR PAIR" $\longrightarrow$ epr is a very entangled state.

$|00\rangle \longrightarrow \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle \longrightarrow \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ $\Rightarrow$ we would redo this computation for $|01\rangle; |10\rangle; |11\rangle$

$|\psi^0\rangle \xrightarrow{H} |\psi^1\rangle \xrightarrow{CNOT} |\psi^2\rangle$

Quantum NAND: QNAND



∴ for $|000\rangle \longrightarrow |001\rangle$ $|001\rangle \longrightarrow |000\rangle$

$|100\rangle \longrightarrow |101\rangle$ $|101\rangle \longrightarrow |100\rangle$

$|010\rangle \longrightarrow |011\rangle$ $|011\rangle \longrightarrow |010\rangle$

$|110\rangle \longrightarrow |110\rangle$ $|111\rangle \longrightarrow |111\rangle$

Quantum Circuits replace with quantum gates, each gate has "out degree" of 1.

Formal Definition:

Universal set in quantum

A quantum circuit on {QNAND, HAD} gates on n+a qubits is a sequence of gates and each gate is H, QNAND

n: #variables; a: Working memory "ancilla"

A function $F: \{0,1\}^n \longrightarrow \{0,m\}^m$ if $n+a \geq m$ is computed by a quantum circuit $P_r[f(x) = (y_1 \cdots y_2)] \geq 0.9$ (Probability)

Thm Shor's Algorithm 1992: there exists an $O(n^2)$ size quantum circuit that can factor n bit numbers.

HW #5 Extra Practice:

Excercise 9.13:

we define the following reduction from haltOnZero.

```
def N(z):
  run Eval(M,0)
  return z
```

we observe that if HALTONZERO(M) outputs 0 then N will not halt and T_{BB} outputs 1.

we observe that if HALTONZERO(M) outputs 1 then N will halt execute z steps and therefore T_{BB} outputs z

2) $L = \{0^m 1 0^n 1 0^{|m-n|} : m,n \geq 0\}$ design CFG

case m=n:
$S \rightarrow 0A01$
$A \rightarrow 0A0 | 1$

case m>n:
$S \rightarrow 0C0$
$C \rightarrow 0C0 | D1$
$D \rightarrow 0D0 | 1$

case: m<n then $0^m 1 0^{m+k} 1 0^k$
$S \rightarrow E0F0$
$E \rightarrow 0E0 | 1$
$F \rightarrow 0F0 | 1$

3a) $H(x) = F(x) \vee G(x)$ I can add an or to my start state

b) $H(x) = F(x) \wedge G(x)$ ⇒ Not intersection

c) $H(x) = NAND(F(x), G(x))$ ⇒ not possible

d). $H(x) = F(x^R)$ where x^R is the reverse it is possible reverse all rules

e). $H(x) = \begin{cases} 1 & x = uv \text{ st. } F(u)=G(v)=1 \\ 0 & \text{otherwise ?} \end{cases}$ ⇒ possible concatenate the start of u and v

f). Not possible

4a) $L = \{x$: 5'th bit from end is 1$\}$

$S \rightarrow A1BCDE$
$A \rightarrow \varepsilon | A1 | A0$
$B \rightarrow 1|0$; $D \rightarrow 1|0$
$C \rightarrow 1|0$; $E \rightarrow 1|0$

b) $L = \{$at least 3 ones: $\{0,1\}^* 1 \{0,1\}^* 1 \{0,1\}^* 1 \{0,1\}^*$

$L = A1A1A1A$
$A \rightarrow \varepsilon | 0A | 1A$

c) $L = \{0^m 1^n : m \neq n\}$

case m>n ; m<n
$S \rightarrow 0A | B1$
$A \rightarrow \varepsilon | 0A | 0A1$
$B \rightarrow \varepsilon | B1 | 0B1$

4d) $L = \{x$: x not in form $0^n 1^n$

$S \rightarrow 0A | B1 | C0 | 1C$
$A \rightarrow \varepsilon | 0A | 0A1$
$B \rightarrow \varepsilon | B1 | 0B1$
$C \rightarrow \varepsilon | 0C | 1C$

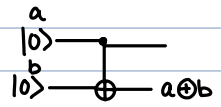
HW #6 extra practice:

1). $|u\rangle = \frac{1}{\sqrt{3}}|0\rangle + \frac{\sqrt{2}}{\sqrt{3}}|1\rangle$ and $|v\rangle = \frac{-\sqrt{2}}{\sqrt{3}}|0\rangle + \frac{1}{\sqrt{3}}|1\rangle$ measure $|+\rangle$

a. $\langle u|+\rangle = \frac{1}{\sqrt{3}}\left(\frac{1}{\sqrt{2}}\right) + \frac{\sqrt{2}}{\sqrt{3}}\left(\frac{1}{\sqrt{2}}\right) = \frac{1}{\sqrt{6}} + \frac{1}{\sqrt{3}} \Rightarrow \frac{1+\sqrt{2}}{\sqrt{6}} \Rightarrow \langle u|+\rangle^2 \Rightarrow \frac{3+2\sqrt{2}}{6}$ probability it is $|u\rangle$

b. $\langle v|-\rangle = \frac{-\sqrt{2}}{\sqrt{3}}\left(\frac{1}{\sqrt{2}}\right) + \frac{1}{\sqrt{6}} \Rightarrow \frac{1}{\sqrt{6}} - \frac{1}{\sqrt{3}} \Rightarrow \frac{1-\sqrt{2}}{\sqrt{6}} \Rightarrow \langle v|+\rangle^2 = \frac{3-2\sqrt{2}}{6}$ probability it is $|v\rangle$

2) Quantum Circuit on two bits for XOR



3) What does the circuit do when passing $\alpha|0\rangle + \beta|1\rangle$ do ? it will in essence output with a 50% $|0\rangle$ and a 50% output 1... more accurately it will be one with probability α^2 and zero with probability $|\beta\rangle^2$

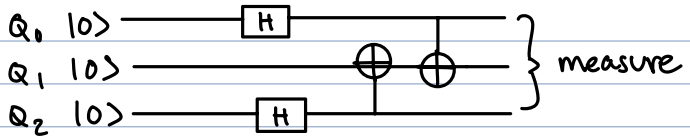
4) $|111\rangle \xrightarrow{H} \frac{1}{\sqrt{2}}|011\rangle - \frac{1}{\sqrt{2}}|111\rangle \xrightarrow{H} \frac{1}{2}|001\rangle - \frac{1}{2}|011\rangle - \frac{1}{2}|101\rangle + \frac{1}{2}|111\rangle \xrightarrow{H} \frac{1}{2\sqrt{2}}\big[|000\rangle - |001\rangle + |011\rangle - |010\rangle - |100\rangle + |101\rangle + |110\rangle - |111\rangle\big]$

5). $\frac{1}{2}|000\rangle + \frac{1}{2}|110\rangle + \frac{1}{2}|011\rangle + \frac{1}{2}|101\rangle$

$|000\rangle \xrightarrow{H \text{ on } 1} \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|100\rangle$

H on 3 $\longrightarrow \frac{1}{2}|000\rangle + \frac{1}{2}|001\rangle + \frac{1}{2}|100\rangle + \frac{1}{2}|101\rangle \rightarrow \frac{1}{2}|000\rangle + \frac{1}{2}|011\rangle + \frac{1}{2}|110\rangle + \frac{1}{2}|101\rangle$

CNOT from 3 to 2 and CNOT from 1 to 2:



3) $\alpha|0\rangle + \beta|1\rangle \xrightarrow{H} \frac{\alpha+\beta}{\sqrt{2}}|0\rangle + \frac{\alpha-\beta}{\sqrt{2}}|1\rangle \xrightarrow{\text{CNOT with } |0\rangle \text{ secon}}$

$\left|\frac{\alpha+\beta}{\sqrt{2}}|0\rangle + \frac{\alpha-\beta}{\sqrt{2}}|1\rangle, \frac{\alpha+\beta}{\sqrt{2}}|0\rangle + \frac{\alpha+\beta}{\sqrt{2}}|1\rangle, \frac{\alpha+\beta}{\sqrt{2}}|0\rangle + \frac{\alpha-\beta}{\sqrt{2}}|1\rangle\right)$

$\left|\frac{\alpha+\beta}{\sqrt{2}}|0\rangle + \frac{\alpha-\beta}{\sqrt{2}}|1\rangle, 0, 0\right\rangle$

ψ

$\alpha|000\rangle + \beta|100\rangle \Rightarrow \frac{\alpha+\beta}{\sqrt{2}}|000\rangle + \frac{\alpha-\beta}{\sqrt{2}}|100\rangle \rightarrow \frac{\alpha+\beta}{\sqrt{2}}|000\rangle + \frac{\alpha-\beta}{\sqrt{2}}|110\rangle$

ψ —
0 —
0 —

$\rightarrow \frac{\alpha+\beta}{\sqrt{2}}|000\rangle + \frac{\alpha-\beta}{\sqrt{2}}|111\rangle$

$\frac{\alpha+\beta}{\sqrt{2}}|00\rangle + \frac{\alpha-\beta}{\sqrt{2}}|11\rangle$

$\left(\frac{\alpha+\beta}{\sqrt{2}}\right)^2$ in $|00\rangle$ and $\left(\frac{\alpha-\beta}{\sqrt{2}}\right)^2$ in $|11\rangle$