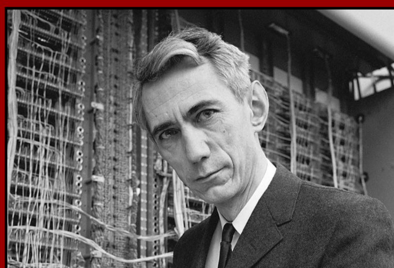
15-251: Great Theoretical Ideas in Computer Science
Fall 2018, Lecture 14

Boolean Formulas and Circuits

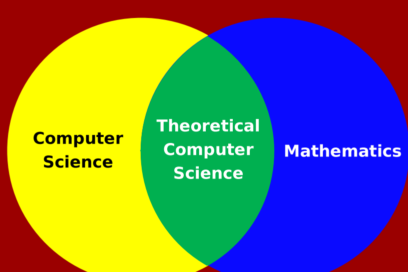


Today

- Briefly mention the "**P** versus **NP**" problem
- Remind you of Boolean formulas
- Tell you about Boolean circuits
- Relate circuit size to algorithmic efficiency
- See why circuits are a good approach to **P** vs. **NP**
- See why circuits are a bad approach to **P** vs. **NP**

P versus **NP**

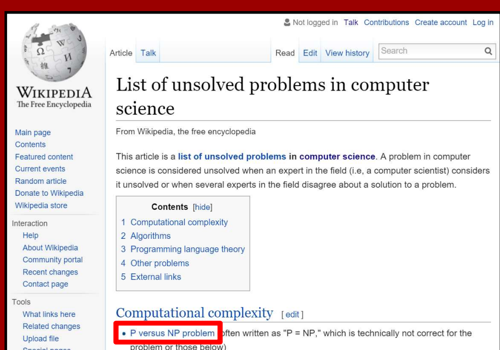
The most famous unsolved problem in Theoretical Computer Science

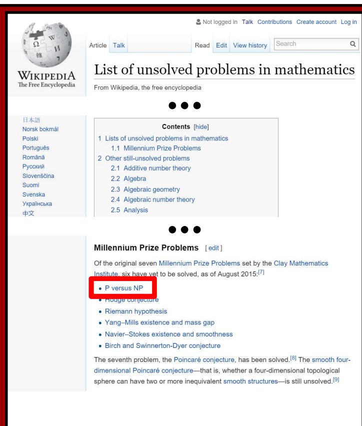


P versus **NP**

The most famous unsolved problem in Theoretical Computer Science

One of the most famous unsolved problems in all of Computer Science, all of Mathematics





P versus **NP**

The most famous unsolved problem in Theoretical Computer Science

One of the most famous unsolved problems in all of Computer Science, all of Mathematics

I can state it for you in ten minutes

Warning: You won't get the full, glorious perspective on why "**P** versus **NP**" is so important until Lectures 15–16

Boolean formulas

You've seen these before in Concepts.

Stuff like this:
$((\neg x \rightarrow y) \wedge ((x \vee z) \leftrightarrow y))$

x, y, z, … Boolean variables, values 0/1 (or T/F)
¬, ∧, ∨, … Boolean connectives (or operations)

A	B	¬A	(A∧B)	(A∨B)	(A→B)	(A↔B)
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

Boolean formulas

You've seen these before in Concepts.

Stuff like this:
$((\neg x \rightarrow y) \wedge ((x \vee z) \leftrightarrow y))$

x, y, z, … Boolean variables, values 0/1 (or T/F)
¬, ∧, ∨, … Boolean connectives (or operations)

Truth assignment: 0/1 value for each variable

A formula is **satisfiable** if there's a truth assignment to the variables making the whole formula true

Truth tables

all possible truth assignments

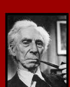
x	y	z	$((\neg x \rightarrow y) \wedge ((x \vee z) \leftrightarrow y))$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

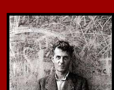
resulting truth value

Satisfiable: At least one 1 in truth table
Unsatisfiable: No 1's in truth table
Tautology: All 1's in truth table

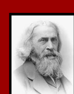
An unsolved problem in Computer Science/Mathematics:

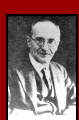
Who invented truth tables?

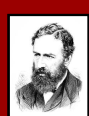

Russell?


Wittgenstein?


Post?


Peirce?


Łukasiewicz?


Jevons?


Ladd–Franklin?

Another unsolved problem in Computer Science/Mathematics:

What is the intrinsic complexity of SAT?

SAT: Given as input a Boolean formula, decide if it is satisfiable or not.

Question: Is SAT decidable?
Answer: Yes.

SAT is decidable

Say the input formula is G.

Brute-Force-Algorithm(G):
Enumerate all truth assignments α.
For each α, compute the truth value it gives G.
If any of them satisfy G, then ACCEPT, else REJECT.

Remark: RAM pseudocode should have some more detail, but I expect you could fill it in.

SAT is decidable

Say the input formula is G.

Brute-Force-Algorithm(G):
Enumerate all truth assignments α.
For each α, compute the truth value it gives G.
If any of them satisfy G, then ACCEPT, else REJECT.

Say the input length (encoding size) of G is N.
Say the # of variables in G is n. (Note: $n \le N$.)

(Although we usually write n for input length, for SAT it's super-traditional to use it for # of variables.)

SAT is decidable

Say the input formula is G.

Brute-Force-Algorithm(G):
Enumerate all truth assignments α.
For each α, compute the truth value it gives G.
If any of them satisfy G, then ACCEPT, else REJECT.

Say the input length (encoding size) of G is N.
Say the # of variables in G is n. (Note: $n \le N$.)

\# of truth assignments? 2^n

∴ running time of **Brute-Force**: $\Omega(2^n)$

SAT is decidable

Say the input formula is G.

Brute-Force-Algorithm(G):
- Enumerate all truth assignments α.
- For each α, compute the truth value it gives G.
- If any of them satisfy G, then ACCEPT, else REJECT.

Say the input length (encoding size) of G is N.
Say the # of variables in G is n. (Note: $n \le N$.)

Running time of **Brute-Force**: $O(2^n \cdot N)$
∴ running time of **Brute-Force**: $\Omega(2^n)$

An unsolved problem in Computer Science/Mathematics

What is the intrinsic complexity of SAT?

SAT: Given as input a Boolean formula, decide if it is satisfiable or not.

We saw SAT is decidable in $O(2^N \cdot N)$ time.

Is SAT decidable in polynomial $O(N^c)$ time?

This is precisely the **P** versus **NP** problem!

The **P** versus **NP** problem

Is SAT decidable in polynomial $O(N^c)$ time?

Warning: You won't get the full, glorious perspective on why "**P** versus **NP**" is so important until Lectures 15–16

The **P** versus **NP** problem

Is SAT decidable in polynomial $O(N^c)$ time?

Most(?) people believe the answer is NO.

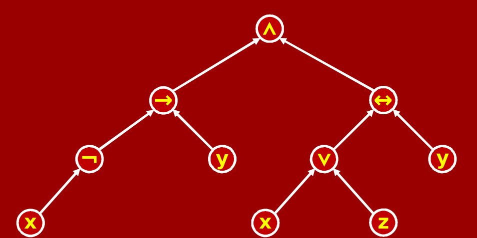
Why is it so hard to prove this?

Polynomial-time algorithms can do
so many amazing, surprising things!

Very hard to prove efficient algorithm don't exist.

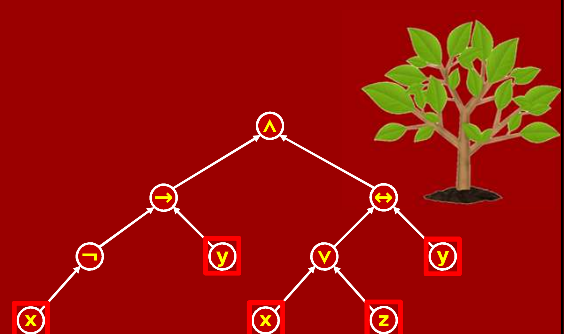
Boolean formulas as binary trees

((¬x → y) ∧ ((x ∨ z) ↔ y))



Boolean formulas as binary trees

Variables at the leaves

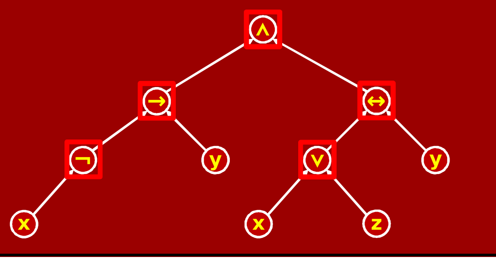


Boolean formulas as binary trees

Variables at the leaves

Connectives at the internal nodes

Connectives have fan-in 2 (except ¬ has fan-in 1)



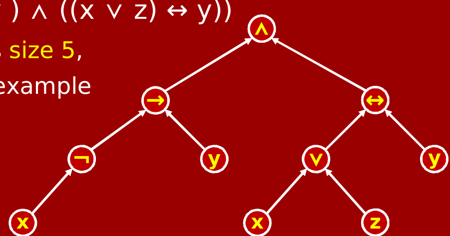
Boolean formula conventions

- The “size” of a formula is the # of leaves (which is also # of variable-appearances).

((¬x → y) ∧ ((x ∨ z) ↔ y))

has size 5,

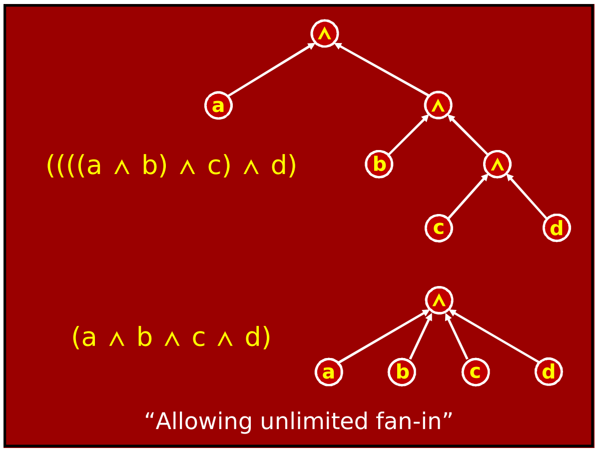
for example



Boolean formula conventions

- The “size” of a formula is the # of leaves (which is also # of variable-appearances).
- Sometimes →, ↔, other connectives allowed. Sometimes just ¬, ∧, ∨. (“De Morgan formulas”) This is “without (much) loss of generality”.
- ((((a ∧ b) ∧ c) ∧ d) ⋯ ∧ z) is often written as (a ∧ b ∧ c ∧ d ∧ ⋯ ∧ z), similarly for ∨.

  Doesn’t affect “size” but does affect “depth”.



More on truth tables

x	y	z	((¬x → y) ∧ ((x ∨ z) ↔ y))
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

all possible truth assignments

resulting truth value

Every n-variable formula yields a truth table.

Two different n-variable formulas can have the same truth table.

More on truth tables

x	y	z	(x ∧ y) ∨ (y ∧ z)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

all possible truth assignments

resulting truth value

Every n-variable formula yields a truth table.

Two different n-variable formulas can have the same truth table.

More on truth tables

all possible truth assignments

x	y	z	(y ∧ (x ∨ z))
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

resulting truth value

Every n-variable formula yields a truth table.

Two different n-variable formulas can have the same truth table.

More on truth tables

all possible truth assignments

x	y	z	(y ∧ (x ∨ z))
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

resulting truth value

If two n-variable formulas have the same truth table, we call them **equivalent**.

More on truth tables

x	y	(x → y)
0	0	1
0	1	1
1	0	0
1	1	1

≡

x	y	(¬x ∨ y)
0	0	1
0	1	1
1	0	0
1	1	1

x	y	(x ∨ y)
0	0	0
0	1	1
1	0	1
1	1	1

≡

x	y	¬(¬x ∧ ¬y)
0	0	0
0	1	1
1	0	1
1	1	1

If two n-variable formulas have the same truth table, we call them **equivalent**.

Boolean functions

We also think of an n-bit truth table as a **Boolean function**, $f : \{0,1\}^n \rightarrow \{0,1\}$.

We think of any formula having that truth table as "computing" that Boolean function.

A Boolean function $f : \{0,1\}^3 \rightarrow \{0,1\}$ can be specified by a truth table. E.g.:

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Or it can be specified by words. E.g.:
"f(x,y,z) = 1 iff at least two input bits are 1"

Question:
How many Boolean functions (truth tables) are there on n variables?

Answer: 2^{2^n}

We know each Boolean formula on n variables "computes" one such function.

Question:
Is every Boolean function (truth table) computed by some Boolean formula?

Is every truth table computed by some formula?

x_1	x_2	x_3	x_4	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

$x_1 \wedge x_2 \wedge x_3 \wedge x_4$

Is every truth table computed by some formula?

x_1	x_2	x_3	x_4	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

$\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4$

Is every truth table computed by some formula?

x_1	x_2	x_3	x_4	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

$x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4$

Is every truth table computed by some formula?

x_1	x_2	x_3	x_4	f
0	0	0	0	**0**
0	0	0	1	**0**
0	0	1	0	**0**
0	0	1	1	**0**
0	1	0	0	**0**
0	1	0	1	**0**
0	1	1	0	**0**
0	1	1	1	**1**
1	0	0	0	**0**
1	0	0	1	**0**
1	0	1	0	**0**
1	0	1	1	**0**
1	1	0	0	**0**
1	1	0	1	**0**
1	1	1	0	**0**
1	1	1	1	**0**

$\neg x_1 \wedge x_2 \wedge x_3 \wedge x_4$

Is every truth table computed by some formula?

x_1	x_2	x_3	x_4	f
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

We can similarly do any truth table with exactly one 1.

Is every truth table computed by some formula?

x_1	x_2	x_3	x_4	f
0	0	0	0	**0**
0	0	0	1	**0**
0	0	1	0	**0**
0	0	1	1	**0**
0	1	0	0	**0**
0	1	0	1	**0**
0	1	1	0	**0**
0	1	1	1	**1**
1	0	0	0	**0**
1	0	0	1	**0**
1	0	1	0	**1**
1	0	1	1	**0**
1	1	0	0	**0**
1	1	0	1	**0**
1	1	1	0	**0**
1	1	1	1	**0**

What if there are **two** 1's?

$(\neg x_1 \wedge x_2 \wedge x_3 \wedge x_4)$
$\vee$
$(x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4)$

Is every truth table computed by some formula?

x_1	x_2	x_3	x_4	f
0	0	0	0	**0**
0	0	0	1	**0**
0	0	1	0	**0**
0	0	1	1	**0**
0	1	0	0	**0**
0	1	0	1	**0**
0	1	1	0	**0**
0	1	1	1	**1**
1	0	0	0	**0**
1	0	0	1	**0**
1	0	1	0	**1**
1	0	1	1	**0**
1	1	0	0	**0**
1	1	0	1	**0**
1	1	1	0	**0**
1	1	1	1	**0**

What if there are **three** 1's?

$(\neg x_1 \wedge x_2 \wedge x_3 \wedge x_4)$ ← (row 0 1 1 1)

$\vee$

$(x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4)$ ← (row 1 0 1 0)

Is every truth table computed by some formula?

x_1	x_2	x_3	x_4	f
0	0	0	0	**0**
0	0	0	1	**0**
0	0	1	0	**0**
0	0	1	1	**0**
0	1	0	0	**0**
0	1	0	1	**0**
0	1	1	0	**0**
0	1	1	1	**1**
1	0	0	0	**0**
1	0	0	1	**0**
1	0	1	0	**1**
1	0	1	1	**0**
1	1	0	0	**0**
1	1	0	1	**0**
1	1	1	0	**1**
1	1	1	1	**0**

What if there are **three** 1's?

$(\neg x_1 \wedge x_2 \wedge x_3 \wedge x_4)$ ← (row 0 1 1 1)

$\vee$

$(x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4)$ ← (row 1 0 1 0)

$\vee$

$(x_1 \wedge x_2 \wedge x_3 \wedge \neg x_4)$ ← (row 1 1 1 0)

We have just done "proof by example" ☺
for the following result (proper proof in Notes):

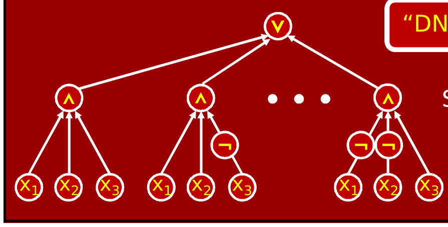
> **Theorem:**
> Every Boolean function (truth table) over n variables can be computed by a formula.

Actually, we missed a case…
…the Boolean function which is always 0.

Well, it's computed by $(x_1 \wedge \neg x_1)$.

Theorem:
Every Boolean function (truth table) over n variables can be computed by a formula.

In fact, by a big ∨ of ∧'s of (possibly negated) variables.



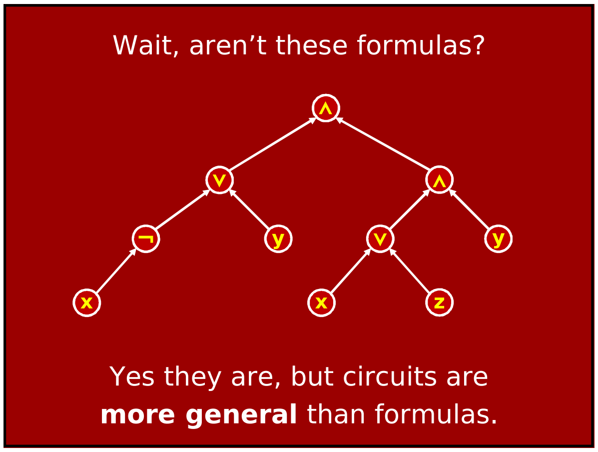
"DNF formula"

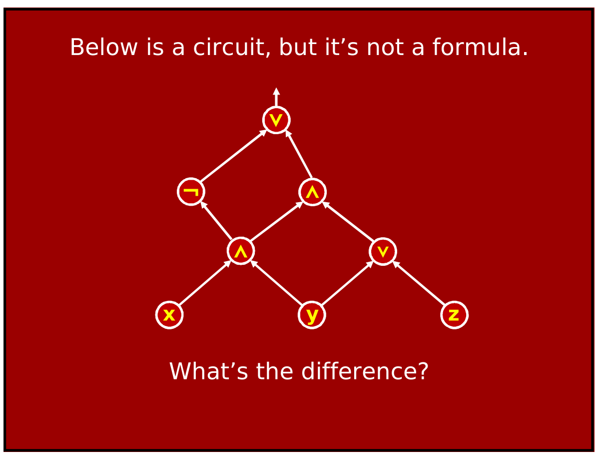
Size ≤ $2^n \cdot n$

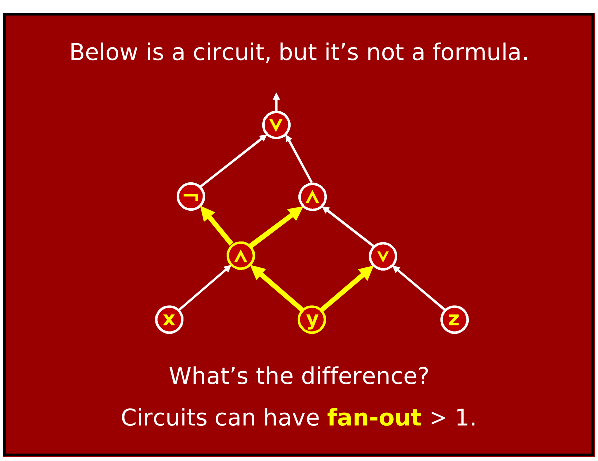
Theorem:
Every Boolean function (truth table) over n variables can be computed by a DNF formula of size ≤ $2^n \cdot n$.

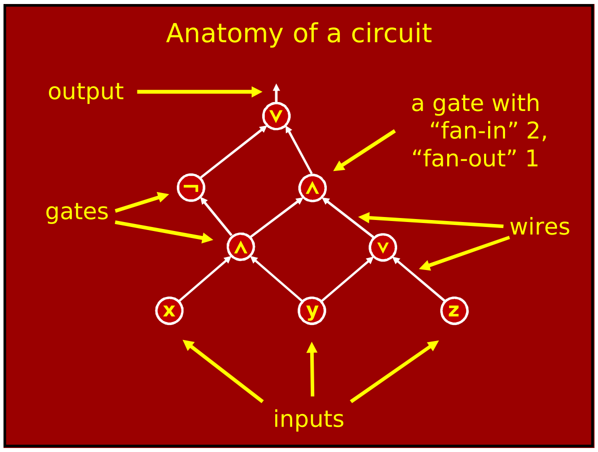
Exercise:
Same statement but with a "CNF formula": a big ∧ of ∨'s of (possibly negated) variables.

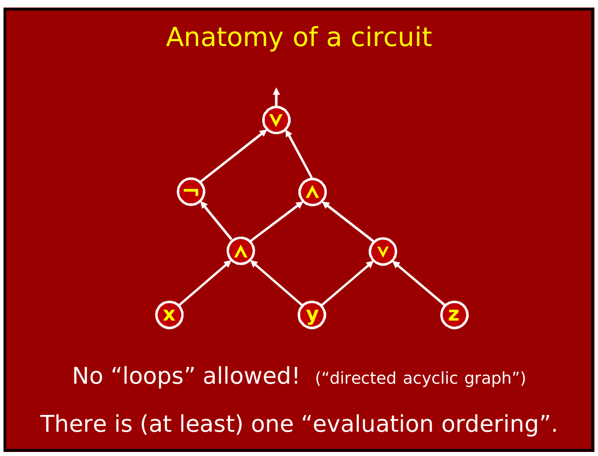
Circuits

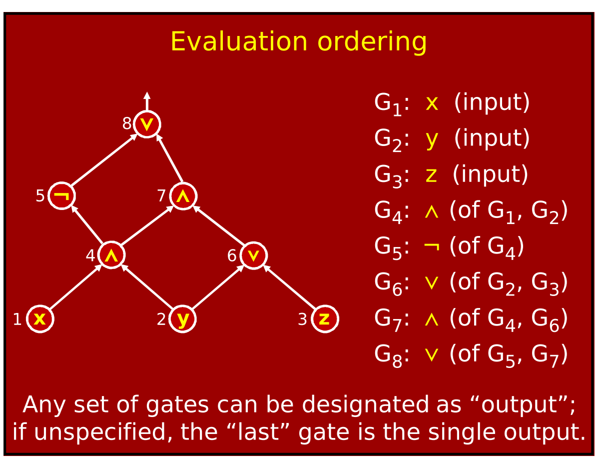

Wait, aren’t these formulas?
∧
∨
∧
¬
y
∨
y
x
x
z
Yes they are, but circuits are
more general than formulas.


Below is a circuit, but it’s not a formula.
∨
¬
∧
∧
∨
x
y
z
What’s the difference?

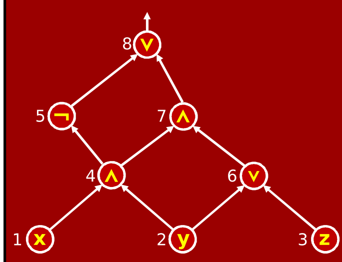

Below is a circuit, but it’s not a formula.
∨
¬
∧
∧
∨
x
y
z
What’s the difference?
Circuits can have **fan-out** > 1.


Anatomy of a circuit
output
a gate with
"fan-in" 2,
"fan-out" 1
gates
wires
∨
¬
∧
∧
∨
x
y
z
inputs


Anatomy of a circuit
∨
¬
∧
∧
∨
x
y
z
No "loops" allowed! ("directed acyclic graph")
There is (at least) one "evaluation ordering".


Evaluation ordering
8
5
7
4
6
1
2
3
G_1: x (input)
G_2: y (input)
G_3: z (input)
G_4: ∧ (of G_1, G_2)
G_5: ¬ (of G_4)
G_6: ∨ (of G_2, G_3)
G_7: ∧ (of G_4, G_6)
G_8: ∨ (of G_5, G_7)
Any set of gates can be designated as "output";
if unspecified, the "last" gate is the single output.

Evaluation ordering



G_1: x (input)
G_2: y (input)
G_3: z (input)
G_4: ∧ (of G_1, G_2)
G_5: ¬ (of G_4)
G_6: ∨ (of G_2, G_3)
G_7: ∧ (of G_4, G_6)
G_8: ∨ (of G_5, G_7)

"Size" of a circuit: # of non-input gates.
(In this example, 5.)

Circuits as programming languages

This is a great way to specify a circuit. →
No picture required!

G_1: x (input)
G_2: y (input)
G_3: z (input)
G_4: ∧ (of G_1, G_2)
G_5: ¬ (of G_4)
G_6: ∨ (of G_2, G_3)
G_7: ∧ (of G_4, G_6)
G_8: ∨ (of G_5, G_7)

Looks like code in a programming language!

Looks like circuit size ≈ running time…

Circuits:

Super-simple.
Looks like a programming language.
Circuit complexity (size) is very concrete.
Circuits can compute any Boolean function.

Why didn't we use circuits
(instead of Turing Machines)
to define computation?!

Good question, we'll come back to that…

Definitional question:
What gates are “allowed” in circuits?

Almost always allowed: $\wedge$ with fan-in 2; $\vee$ with fan-in 2; $\neg$ with fan-in 1

Usually allowed: 0 with fan-in 0; 1 with fan-in 0

Sometimes allowed: any fan-in 2 gate; e.g., $\equiv$ (equals), $\oplus$ (XOR)

Often allowed: $\wedge$ with **any** fan-in; $\vee$ with **any** fan-in

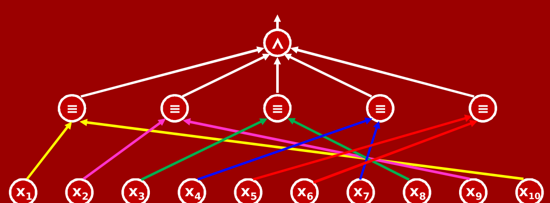
Doesn’t make a big difference, but always ask.

Let’s build a circuit for 10-bit PALINDROMES

$f : \{0,1\}^{10} \to \{0,1\}$

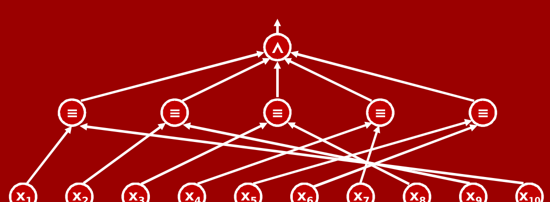
$f(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,x_9,x_{10}) = 1$
if and only if input string is same as its reverse

Let’s be liberal, allow all gates on previous slide.



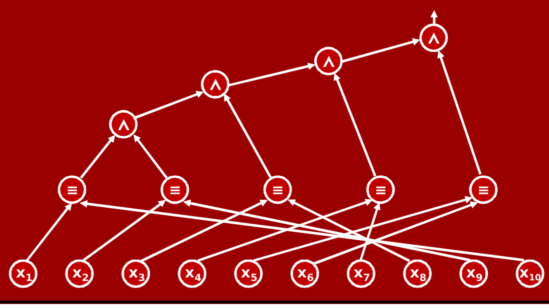
Size? 6 (Depth? 2)

What if we only allow fan-in 2 gates?



Size? 9 (Depth? higher)

What if we only allow fan-in 2 gates?

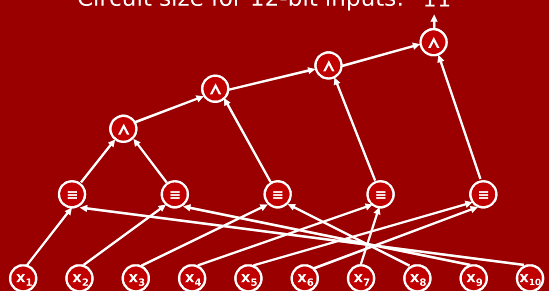


Circuit size for 10-bit inputs: 9

Circuit size for 11-bit inputs: 9

Circuit size for 12-bit inputs: 11

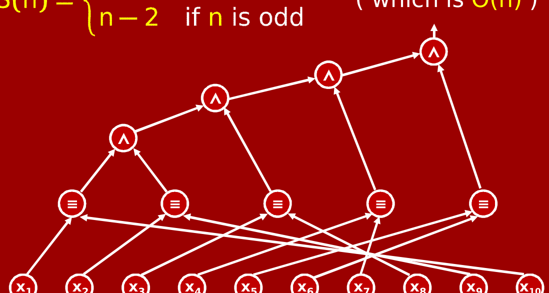
Circuit size for 12-bit inputs: 11



Continuing this pattern, we can get a circuit deciding n-bit inputs for PALINDROME having size...

$$S(n) = \begin{cases} n-1 & \text{if } n \text{ is even} \\ n-2 & \text{if } n \text{ is odd} \end{cases}$$

(which is $\Theta(n)$)



Circuits:

Super-simple.
Look like a programming language.
Circuit complexity (size) is very concrete.
Circuits can compute any Boolean function.

Why didn’t we use circuits
(instead of Turing Machines)
to define computation?!

From Lecture 2…

input → “computer” → output

What is **computation**?
What is an **algorithm**?

How can we mathematically define them?

Inspirational quotation

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

Stephen Kleene

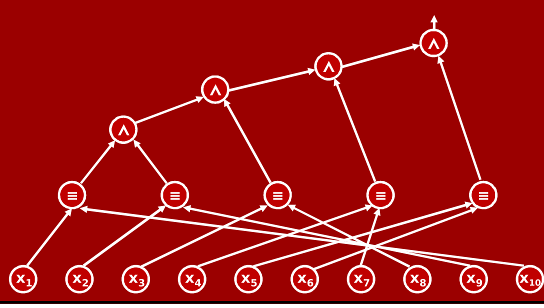
“Circuits are an **infinite** answer to an infinite number of questions ☹”

me

Consider the language PALINDROMES ⊆ $\{0,1\}^*$

How can we compute it using circuits?

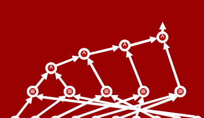
Well, for length-10 inputs we had something…



Consider the language PALINDROMES ⊆ $\{0,1\}^*$

How can we compute it using circuits?

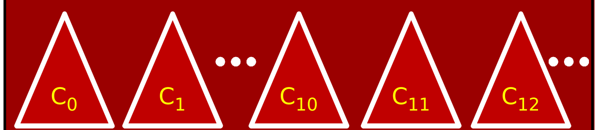
Well, for length-10 inputs we had something…

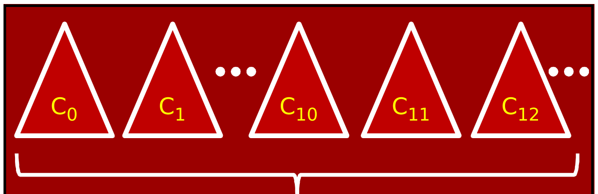


Consider the language PALINDROMES ⊆ $\{0,1\}^*$

How can we compute it using circuits?

Well, for length-10 inputs we had something…
For length-11 inputs we had something else…
For length-12 inputs we had something else…
For length-0 inputs we had something else…
For length-1 inputs we had something else…

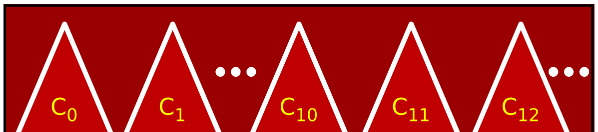




This is called a "family of circuits".

It's a fine mathematical concept, but we don't like it to use it to define "computation", because it's **infinite**.

(It sort of begs the question: In real life, how do you get C_n? Probably there's an **algorithm** that on input n, outputs C_n…)



Definition:

A family of circuits $\mathcal{C}$ is an infinite sequence $C_0, C_1, C_2, \ldots$ where C_n is a circuit with n inputs.

We say $\mathcal{C}$ decides $L \subseteq \{0,1\}^*$ if for all $n \in \mathbb{N}$, C_n decides $L_n = L \cap \{0,1\}^n$.

The size of $\mathcal{C}$ is the function $S : \mathbb{N} \to \mathbb{N}$ defined by $S(n) =$ size of C_n.

Example

Let $\mathcal{C}$ be the family of circuits where C_n is…

Then $\mathcal{C}$ decides the language PALINDROMES and has size O(n); more precisely,

$$S(n) = \begin{cases} n-1 & \text{if } n \text{ is even} \\ n-2 & \text{if } n \text{ is odd} \end{cases}$$

*only for n ≥ 4; special cases for n=0,1,2,3

G_1:	x_1	(input)
G_2:	x_2	(input)
	…	
G_n:	x_n	(input)
E_1:	≡	(of G_1, G_n)
E_2	≡	(of G_2, G_{n-1})
E_3:	≡	(of G_3, G_{n-2})
	…	
$E_{\lfloor n/2\rfloor}$:	≡	(of $G_{\lfloor n/2\rfloor}$, $G_{\lceil n/2\rceil+1}$)
A_1:	∧	(of E_1, E_2)
A_2:	∧	(of A_1, E_3)
A_3:	∧	(of A_2, E_4)
	…	
$A_{\lfloor n/2\rfloor-1}$:	∧	(of $A_{\lfloor n/2\rfloor-1}$, $E_{\lfloor n/2\rfloor}$)

Recall: Every n-bit Boolean function computable by a formula/circuit of size $O(2^n \cdot n)$.

(I don't mean to alarm you, but this includes HALT!!)

Consequence:

Every language is computed by a family of circuits of size $O(2^n \cdot n)$.

Recall: Every n-bit Boolean function computable by a formula/circuit of size $O(2^n \cdot n)$.

Easy improvement:

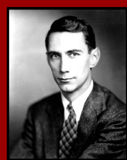
Every language is computed by a family of circuits of size $O(2^n)$.

Recall: Every n-bit Boolean function computable by a formula/circuit of size $O(2^n \cdot n)$.

Slightly trickier improvement:

Every language is computed by a family of circuits of size $O(2^n/n)$.

Proved by the great Claude Shannon in 1949.



TM time versus circuit size

Theorem:

Suppose there is a TM deciding L in time $T(n)$.
Then it can be converted into a circuit family deciding L with size $S(n) = O(T(n)^2)$.

If you like a challenge, try to prove this yourself.

If you don't like a challenge, but are still curious, see the Notes online.

If you neither like a challenge nor are curious, ☹.

We'll need theorem when studying "NP-hardness."

TM time versus circuit size

Theorem:

Suppose there is a TM deciding L in time $T(n)$.
Then it can be converted into a circuit family deciding L with size $S(n) = O(T(n)^2)$.

Corollary:

Any L solvable in polynomial time on TMs (or in RAM model) has polynomial-size circuits.

TM time versus circuit size

Corollary:

If you want to show some L is **not** solvable in polynomial time, suffices to show it is **not** solvable by polynomial-size circuit families.

Corollary:

Any L solvable in polynomial time on TMs (or in RAM model) has polynomial-size circuits.

TM time versus circuit size

Corollary:

If you want to show some L is **not** solvable in polynomial time, suffices to show it is **not** solvable by polynomial-size circuit families.

In the '80s, this was viewed as the approach that would solve **P** versus **NP**.

"Just" have to show that SAT doesn't have polynomial-size circuit families.

Shannon's Theorem 1:

Every n-bit Boolean function has an $\wedge/\vee/\neg$ circuit of size $O(2^n/n)$

Shannon's Theorem 2:

Almost every n-bit Boolean function **requires** a circuit of size $\Omega(2^n/n)$

(even when all fan-in 2 gates are allowed)

"Essentially all computational problems require exponential circuit complexity."

Shannon's Theorem 2:

Almost every n-bit Boolean function **requires** a circuit of size $\Omega(2^n/n)$.

Proof:

Let $s = (1/4)\, 2^n/n$. We'll show: There are $\leq (1.5)^{2^n}$ circuits of size s.
But there are **way** more n-bit Boolean functions: 2^{2^n}.

Think of the "programming language" form of a size-s circuit.
After the n input gates, we have s more lines. Each defined by a gate type (16 choices) and two previous lines ($\leq n+s$ choices).
So there are at most $[16 \cdot (n+s) \cdot (n+s)]^s$ possible circuits.
The $[\cdots]$ quantity is $\leq 64s^2$ because $n+s \leq 2s$, and $64s^2 \leq (2^n)^2$ for large n. So there are at most $[(2^n)^2]^s = 2^{2ns} = 2^{(1/2)\, 2^n} = (1.41\ldots)^{2^n}$ size-s circuits and most n-bit Boolean functions need a larger size.

Shannon's Theorem 2:

Almost every n-bit Boolean function **requires** a circuit of size $\Omega(2^n/n)$.

"Essentially all computational problems require exponential circuit complexity."

So… what's an example of one?

If SAT is an example, we resolve **P** versus **NP**!

Or… can we just find **any** explicit example?!

Challenge: Find an explicit n-bit function requiring large circuit size.

Shannon: Practically **all** functions need $\Omega(2^n/n)$.

1965: Kloss & Malyshev show a certain simple function requires size $\geq 2n - 3$

1977: Paul & Stockmeyer show certain simple functions requires size $\geq 2.5n - 1.5$

1984: N. Blum showed a certain pretty simply function requires size $\geq 3n - 3$

Consider n = 50

Shannon: Practically **all** functions 20 trillion

1965: Kloss & Malyshev show a certain simple function requires size $\geq 2n - 3$

1977: Paul & Stockmeyer show certain simple functions requires size $\geq 2.5n - 1.5$

1984: N. Blum showed a certain pretty simple function requires size $\geq$ 147

1965: Kloss & Malyshev show a certain simple function requires size $\geq 2n - 3$

1977: Paul & Stockmeyer show certain simple functions requires size $\geq 2.5n - 1.5$

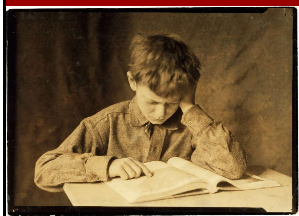
1984: N. Blum showed a certain pretty simple function requires size $\geq 3n - 3$

Good news!!

2016: Find, Golevnev, Hirsch, Kulikov showed a certain function requires size $\geq (3+1/86)n - O(n^{.8})$

This pretty much sums up where we are on **P** versus **NP**.

Study Guide



Definitions:

- Boolean formulas
- Truth tables
- Boolean functions
- The SAT problem
- Circuits
- Circuit familes & size

Theorems:

- Every function can be computed by a DNF
- Almost every function requires circuits of size $\Omega(2^n/n)$.