Theory of Computation
Complexity classes, P, EXP

Complexity classes

Complexity classes

- In other CS classes, we might ask what problems can we solve in a particular runtime (e.g. $O(n)$, $O(n^2)$, etc.)

Complexity classes

- In other CS classes, we might ask what problems can we solve in a particular runtime (e.g. $O(n)$, $O(n^2)$, etc.)
- In this class, we are more interested in coarser classifications

Complexity classes

- In other CS classes, we might ask what problems can we solve in a particular runtime (e.g. $O(n)$, $O(n^2)$, etc.)
- In this class, we are more interested in coarser classifications
  - what problems require the same "level/tier" of resources

Complexity classes

- In other CS classes, we might ask what problems can we solve in a particular runtime (e.g. $O(n)$, $O(n^2)$, etc.)
- In this class, we are more interested in coarser classifications
  - what problems require the same "level/tier" of resources
  - Which problems can be solved "efficiently"?

Complexity classes

- In other CS classes, we might ask what problems can we solve in a particular runtime (e.g. $O(n)$, $O(n^2)$, etc.)
- In this class, we are more interested in coarser classifications
  - what problems require the same "level/tier" of resources
  - Which problems can be solved "efficiently"?
  - **What problems can't be solved efficiently?**

Complexity classes

- **Recall:** a language is a set of strings

Complexity classes

- **Recall:** a language is a set of strings
- **Def:** a **complexity class** is a set of *languages*

Complexity classes

- **Recall:** a language is a set of strings
- **Def:** a **complexity class** is a set of *languages*
- We have already seen some complexity classes:

Complexity classes

- **Recall:** a language is a set of strings
- **Def:** a **complexity class** is a set of *languages*
- We have already seen some complexity classes:
  - REG: the regular languages

Complexity classes

- **Recall:** a language is a set of strings
- **Def:** a **complexity class** is a set of *languages*
- We have already seen some complexity classes:
  - REG: the regular languages
  - D: the decidable languages

Complexity classes

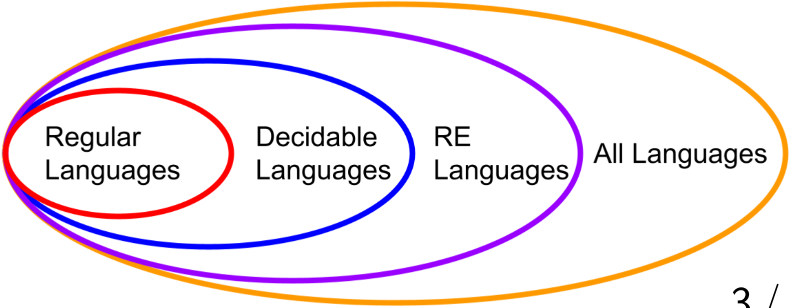
- **Recall:** a language is a set of strings
- **Def:** a **complexity class** is a set of *languages*
- We have already seen some complexity classes:
  - REG: the regular languages
  - D: the decidable languages
  - RE: the recursively enumerable languages

Complexity classes

- **Recall:** a language is a set of strings
- **Def:** a **complexity class** is a set of *languages*
- We have already seen some complexity classes:
  - REG: the regular languages
  - D: the decidable languages
  - RE: the recursively enumerable languages
- Some of these classes are bigger than others!

Complexity classes

- **Recall:** a language is a set of strings
- **Def:** a **complexity class** is a set of *languages*
- We have already seen some complexity classes:
  - REG: the regular languages
  - D: the decidable languages
  - RE: the recursively enumerable languages
- Some of these classes are bigger than others!



TIME-based complexity classes

TIME-based complexity classes

- Let $T : \mathbb{N} \to \mathbb{N}$ be a runtime function

TIME-based complexity classes

- Let $T : \mathbb{N} \to \mathbb{N}$ be a runtime function
- **Def:** The class $\mathrm{TIME}(T(n))$ is the set of all languages that can be decided by a machine that runs in $O(T(n))$ time

TIME-based complexity classes

- Let $T : \mathbb{N} \to \mathbb{N}$ be a runtime function
- **Def:** The class $\mathrm{TIME}(T(n))$ is the set of all languages that can be decided by a machine that runs in $O(T(n))$ time
- The language $L = \{0^k 1^k | k \geq 0\} \in \mathrm{TIME}(n^2)$

TIME-based complexity classes

- Let $T : \mathbb{N} \rightarrow \mathbb{N}$ be a runtime function
- **Def:** The class $\mathrm{TIME}(T(n))$ is the set of all languages that can be decided by a machine that runs in $O(T(n))$ time
- The language $L = \{0^k 1^k | k \geq 0\} \in \mathrm{TIME}(n^2)$
  - In fact, $L \in \mathrm{TIME}(n \log(n))$ - see Sipser

The class P

The class P

- We want a working definition what it means for a problem to be solved "efficiently"

The class P

- We want a working definition what it means for a problem to be solved "efficiently"
- **Def:** The class P is the set of all languages that can be decided in polynomial time

The class P

- We want a working definition what it means for a problem to be solved "efficiently"
- **Def:** The class P is the set of all languages that can be decided in polynomial time
  - $O(n^c)$ for some constant c

The class P

- We want a working definition what it means for a problem to be solved "efficiently"
- **Def:** The class P is the set of all languages that can be decided in polynomial time
  - $O(n^c)$ for some constant c
- Alternate definition:

$$\mathrm{P} = \bigcup_{c} \mathrm{TIME}(T(n^c))$$

The class P

- We want a working definition what it means for a problem to be solved "efficiently"
- **Def:** The class P is the set of all languages that can be decided in polynomial time
  - $O(n^c)$ for some constant c
- Alternate definition:

$$\mathrm{P} = \bigcup_c \mathrm{TIME}(T(n^c))$$

- In this course, we will use P as a proxy for "tractable" problems

Length of numeric inputs

Length of numeric inputs

- The numeric value of a number isn't the same as the length of its encoding!

Length of numeric inputs

- The numeric value of a number isn't the same as the length of its encoding!
- Let's consider the number $n = 16$

Length of numeric inputs

- The numeric value of a number isn't the same as the length of its encoding!
- Let's consider the number $n = 16$
- **Unary encoding**: $\langle 16 \rangle = \underbrace{1111111111111111}_{|\langle n \rangle| \in O(n)}$

Length of numeric inputs

- The numeric value of a number isn't the same as the length of its encoding!
- Let's consider the number $n = 16$
- **Unary encoding**: $\langle 16 \rangle = \underbrace{1111111111111111}_{|\langle n \rangle| \in O(n)}$
- **Binary encoding**: $\langle 16 \rangle = \underbrace{10000}_{\substack{|\langle n \rangle| \in O(\log(n)) \\ n \in O(2^{|\langle n \rangle|})}}$

Length of numeric inputs

- The numeric value of a number isn't the same as the length of its encoding!
- Let's consider the number $n = 16$
- **Unary encoding**: $\langle 16 \rangle = \underbrace{1111111111111111}_{|\langle n \rangle| \in O(n)}$
- **Binary encoding**: $\langle 16 \rangle = \underbrace{10000}_{\substack{|\langle n \rangle| \in O(\log(n)) \\ n \in O(2^{|\langle n \rangle|})}}$
- An 8-byte unary integer cannot represent numbers bigger than 32!!!

Length of numeric inputs

- The numeric value of a number isn't the same as the length of its encoding!
- Let's consider the number $n = 16$
- **Unary encoding**: $\langle 16 \rangle = \underbrace{1111111111111111}_{|\langle n \rangle| \in O(n)}$
- **Binary encoding**: $\langle 16 \rangle = \underbrace{10000}_{\substack{|\langle n \rangle| \in O(\log(n)) \\ n \in O(2^{|\langle n \rangle|})}}$
- An 8-byte unary integer cannot represent numbers bigger than 32!!!
- If the input is in binary (or base 10 or base 16), we have to be careful about runtime analysis

Runtime with numeric inputs

What is the running time of this algorithm?

1. Receive a number $\langle N \rangle$ as input in binary
2. For $i = 2...(N-1)$:
   2.1 If $N \% i == 0$, immediately reject
3. If we finish the loop, accept

Runtime with numeric inputs

What is the running time of this algorithm?

1. Receive a number $\langle N \rangle$ as input in binary
2. For $i = 2...(N-1)$:
   2.1 If $N \% i == 0$, immediately reject
3. If we finish the loop, accept

- $O(N)$ loop iterations

Runtime with numeric inputs

What is the running time of this algorithm?

1. Receive a number $\langle N \rangle$ as input in binary
2. For $i = 2...(N-1)$:
   2.1 If $N \; \% \; i == 0$, immediately reject
3. If we finish the loop, accept

- $O(N)$ loop iterations
- $|\langle N \rangle| = O(\log(N))$

Runtime with numeric inputs

What is the running time of this algorithm?

1. Receive a number $\langle N \rangle$ as input in binary
2. For $i = 2...(N-1)$:
   2.1 If $N \% i == 0$, immediately reject
3. If we finish the loop, accept

- $O(N)$ loop iterations
- $|\langle N \rangle| = O(\log(N))$
- $N = 2^{|\langle N \rangle|}$

Runtime with numeric inputs

What is the running time of this algorithm?

1. Receive a number $\langle N \rangle$ as input in binary
2. For $i = 2...(N-1)$:
   2.1 If $N \ \% \ i == 0$, immediately reject
3. If we finish the loop, accept

- $O(N)$ loop iterations
- $|\langle N \rangle| = O(\log(N))$
- $N = 2^{|\langle N \rangle|}$
- $O(2^{|\langle N \rangle|})$ loop iterations!!!

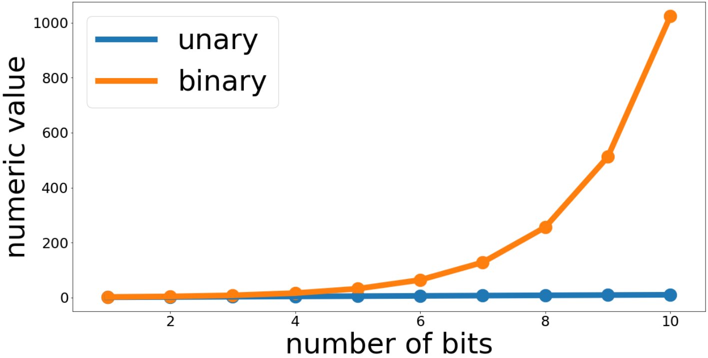
Runtime with numeric inputs

What is the running time of this algorithm?

1. Receive a number $\langle N \rangle$ as input in binary
2. For $i = 2 \ldots (N-1)$:
   2.1 If $N \ \% \ i == 0$, immediately reject
3. If we finish the loop, accept

- $O(N)$ loop iterations
- $|\langle N \rangle| = O(\log(N))$
- $N = 2^{|\langle N \rangle|}$
- $O(2^{|\langle N \rangle|})$ loop iterations!!!
- This is exponential in the length of the input!!!

Runtime with numeric inputs



The language COPRIMES

The language COPRIMES

$$\mathrm{COPRIMES} = \{\langle x, y\rangle \,|\, \gcd(x, y) = 1\}$$

- We receive two binary numbers as input

The language COPRIMES

$$\text{COPRIMES} = \{\langle x, y\rangle \mid \text{gcd}(x, y) = 1\}$$

- We receive two binary numbers as input
- We want to check if they have any common factors (besides 1)

The language $\mathrm{COPRIMES}$

$$\mathrm{COPRIMES} = \{\langle x, y\rangle \mid \mathsf{gcd}(x, y) = 1\}$$

- We receive two binary numbers as input
- We want to check if they have any common factors (besides 1)
- **Naive approach:** for $i = 1, \ldots, \min(x, y)$, check if i is a common factor, and output the maximum common factor found

The language COPRIMES

$$\text{COPRIMES} = \{\langle x, y \rangle \mid \text{gcd}(x, y) = 1\}$$

- We receive two binary numbers as input
- We want to check if they have any common factors (besides 1)
- **Naive approach:** for $i = 1, \ldots, \min(x, y)$, check if i is a common factor, and output the maximum common factor found
- This is $O(n)$ in the *value* of x and y...

The language COPRIMES

$$\mathrm{COPRIMES} = \{\langle x, y \rangle \mid \gcd(x, y) = 1\}$$

- We receive two binary numbers as input
- We want to check if they have any common factors (besides 1)
- **Naive approach:** for $i = 1, \ldots, \min(x, y)$, check if i is a common factor, and output the maximum common factor found
- This is $O(n)$ in the *value* of x and y...
- ...which is $O(2^n)$ in the *length* of $\langle x, y \rangle$

COPRIMES $\in$ P

COPRIMES $\in$ P

We will use the **Euclidean Algorithm** – possibly the oldest recorded algorithm

COPRIMES $\in$ P

We will use the **Euclidean Algorithm** – possibly the oldest recorded algorithm

1. If $x < y$, swap x and y

COPRIMES $\in$ P

We will use the **Euclidean Algorithm** – possibly the oldest recorded algorithm

1. If $x < y$, swap x and y
2. Repeat until $y = 0$:

COPRIMES $\in$ P

We will use the **Euclidean Algorithm** – possibly the oldest recorded algorithm

1. If $x < y$, swap x and y
2. Repeat until $y = 0$:
   2.1 $x \leftarrow x \% y$

COPRIMES $\in$ P

We will use the **Euclidean Algorithm** – possibly the oldest recorded algorithm

1. If $x < y$, swap x and y
2. Repeat until $y = 0$:
   2.1 $x \leftarrow x \,\%\, y$
   2.2 Swap x and y

COPRIMES $\in$ P

We will use the **Euclidean Algorithm** – possibly the oldest recorded algorithm

1. If $x < y$, swap x and y
2. Repeat until $y = 0$:
   2.1 $x \leftarrow x \% y$
   2.2 Swap x and y
3. If $x = 1$, accept $\langle x, y \rangle$; otherwise reject

COPRIMES $\in$ P

78 ÷ 66 = 1 remainder 12 $\quad (78 = 66 \times 1 + 12)$

66 ÷ 12 = 5 remainder 6 $\quad (66 = 12 \times 5 + 6)$

12 ÷ 6 = 2 remainder 0 $\quad (12 = 6 \times 2 + 0)$

6 = Greatest Common Factor

COPRIMES $\in$ P

We will use the **Euclidean Algorithm** – possibly the oldest recorded algorithm

1. If $x < y$, swap x and y
2. Repeat until $y = 0$:
   2.1 $x \leftarrow x \,\%\, y$
   2.2 Swap x and y
3. If $x = 1$, accept $\langle x, y \rangle$; otherwise reject

Claim: This step cuts x in half

COPRIMES $\in$ P

We will use the **Euclidean Algorithm** – possibly the oldest recorded algorithm

1. If $x < y$, swap x and y
2. Repeat until $y = 0$:
   2.1 $x \leftarrow x \,\%\, y$
   2.2 Swap x and y
3. If $x = 1$, accept $\langle x, y \rangle$; otherwise reject

Claim: This step cuts x in half

- **Case 1:** $y \leq \frac{x}{2}$. Then $x \,\%\, y < y \leq \frac{x}{2}$

COPRIMES $\in$ P

We will use the **Euclidean Algorithm** – possibly the oldest recorded algorithm

1. If $x < y$, swap x and y
2. Repeat until $y = 0$:
   2.1 $x \leftarrow x \,\%\, y$
   2.2 Swap x and y
3. If $x = 1$, accept $\langle x, y \rangle$; otherwise reject

Claim: This step cuts x in half

- **Case 1:** $y \leq \frac{x}{2}$. Then $x \,\%\, y < y \leq \frac{x}{2}$
- **Case 2:** $y > \frac{x}{2}$. Then $x \,\%\, y = x - y < \frac{x}{2}$

COPRIMES $\in$ P

We will use the **Euclidean Algorithm** – possibly the oldest recorded algorithm

1. If $x < y$, swap x and y
2. Repeat until $y = 0$:
   2.1 $x \leftarrow x \mathbin{\%} y$
   2.2 Swap x and y
3. If $x = 1$, accept $\langle x, y \rangle$; otherwise reject

Claim: There are $O(n = |\langle x, y \rangle|)$ loop iterations

COPRIMES $\in$ P

We will use the **Euclidean Algorithm** – possibly the oldest recorded algorithm

1. If $x < y$, swap x and y
2. Repeat until $y = 0$:
   2.1 $x \leftarrow x \% y$
   2.2 Swap x and y
3. If $x = 1$, accept $\langle x, y \rangle$; otherwise reject

Claim: There are $O(n = |\langle x, y \rangle|)$ loop iterations

- After two iterations, both x and y have been cut in half

COPRIMES $\in$ P

We will use the **Euclidean Algorithm** – possibly the oldest recorded algorithm

1. If $x < y$, swap x and y
2. Repeat until $y = 0$:
   2.1 $x \leftarrow x \ \% \ y$
   2.2 Swap x and y
3. If $x = 1$, accept $\langle x, y \rangle$; otherwise reject

Claim: There are $O(n = |\langle x, y \rangle|)$ loop iterations

- After two iterations, both x and y have been cut in half
- The number of times we can cut the input in half is $\log(\max\{x, y\}) = O(|\langle x, y \rangle|)$

COPRIMES $\in$ P

We will use the **Euclidean Algorithm** – possibly the oldest recorded algorithm

1. If $x < y$, swap x and y
2. Repeat until $y = 0$:
   2.1 $x \leftarrow x \,\%\, y$
   2.2 Swap x and y
3. If $x = 1$, accept $\langle x, y \rangle$; otherwise reject

- Modular reduction (and other arithmetic) can be calculated in polynomial time

COPRIMES $\in \mathrm{P}$

We will use the **Euclidean Algorithm** – possibly the oldest recorded algorithm

1. If $x < y$, swap x and y
2. Repeat until $y = 0$:
   2.1 $x \leftarrow x \,\%\, y$
   2.2 Swap x and y
3. If $x = 1$, accept $\langle x, y \rangle$; otherwise reject

- Modular reduction (and other arithmetic) can be calculated in polynomial time
- $O(n)$ loop iterations $\times O(n^c)$ steps per loop iteration $= O(n^c) \in \mathrm{P}$

The language UNARY-SUBSET-SUM

$$\text{UNARY-SUBSET-SUM} = \left\{ \langle B | x_1, x_2, \ldots x_n \rangle \middle| \begin{array}{c} \text{B is unary} \\ \text{there is a combination of } x_i \text{ (no repeats)} \\ \text{that add up to B} \end{array} \right\}$$

The language UNARY-SUBSET-SUM

$$\text{UNARY-SUBSET-SUM} = \left\{ \langle B | x_1, x_2, \ldots x_n \rangle \middle| \begin{array}{c} \text{B is unary} \\ \text{there is a combination of } x_i \text{ (no repeats)} \\ \text{that add up to B} \end{array} \right\}$$

Example: $\langle 31 | 7, 4, 9, 5, 20 \rangle$
Solution: $7 + 4 + 20 = 31 \checkmark$

The language UNARY-SUBSET-SUM

$$\text{UNARY-SUBSET-SUM} = \left\{ \langle B | x_1, x_2, \ldots x_n \rangle \middle| \begin{array}{c} \text{B is unary} \\ \text{there is a combination of } x_i \text{ (no repeats)} \\ \text{that add up to B} \end{array} \right\}$$

Example: $\langle 31 | 7, 4, 9, 5, 20 \rangle$
Solution: $7 + 4 + 20 = 31$✓

Example: $\langle 101 | 6, 8, 10 \rangle$
Solution: It is impossible; $6 + 8 + 10 = 24 < 101$

The language UNARY-SUBSET-SUM

Which of the following sets are part of UNARY-SUBSET-SUM?

A. $\langle 0 | 1, 2, 3, 4, 5 \rangle$
B. $\langle 13 | 3, 3, 3 \rangle$
C. $\langle 40 | 13, 26, 15, 24 \rangle$
D. $\langle 45 | 2, 3, 10, 17, 30 \rangle$

The language UNARY-SUBSET-SUM

Which of the following sets are part of UNARY-SUBSET-SUM?

A. $\langle 0 | 1, 2, 3, 4, 5 \rangle$ ✓
B. $\langle 13 | 3, 3, 3 \rangle$
C. $\langle 40 | 13, 26, 15, 24 \rangle$
D. $\langle 45 | 2, 3, 10, 17, 30 \rangle$ ✓

UNARY-SUBSET-SUM $\in$ P

UNARY-SUBSET-SUM $\in$ P

Technique: dynamic programming

UNARY-SUBSET-SUM $\in$ P

Technique: dynamic programming

1. $A \leftarrow (n+1) \times (B+1)$ matrix.

UNARY-SUBSET-SUM $\in$ P

Technique: dynamic programming

1. $A \leftarrow (n+1) \times (B+1)$ matrix.
2. Initialize $A[i, 0]$ to TRUE for all i; Initialize all other elements to FALSE

UNARY-SUBSET-SUM $\in$ P

Technique: dynamic programming

1. $A \leftarrow (n+1) \times (B+1)$ matrix.
2. Initialize $A[i, 0]$ to TRUE for all i; Initialize all other elements to FALSE
3. For $i = 1 \ldots n$:

UNARY-SUBSET-SUM $\in$ P

Technique: dynamic programming

1. $A \leftarrow (n+1) \times (B+1)$ matrix.
2. Initialize $A[i, 0]$ to TRUE for all i; Initialize all other elements to FALSE
3. For $i = 1 \ldots n$:
   3.1 For $j = 1 \ldots B$:

UNARY-SUBSET-SUM $\in$ P

Technique: dynamic programming

1. $A \leftarrow (n+1) \times (B+1)$ matrix.
2. Initialize $A[i, 0]$ to TRUE for all i; Initialize all other elements to FALSE
3. For $i = 1 \ldots n$:
   3.1 For $j = 1 \ldots B$:
      3.1.1 If $A[i-1, j] = \mathrm{TRUE}$, or if $j \geq x_i$ and $A[i-1, j-x_i] = \mathrm{TRUE}$, set $A[i]$ to TRUE

UNARY-SUBSET-SUM $\in$ P

Technique: dynamic programming

1. $A \leftarrow (n+1) \times (B+1)$ matrix.
2. Initialize $A[i,0]$ to TRUE for all i; Initialize all other elements to FALSE
3. For $i = 1 \ldots n$:
   3.1 For $j = 1 \ldots B$:
      3.1.1 If $A[i-1,j] = \mathrm{TRUE}$, or if $j \geq x_i$ and $A[i-1, j-x_i] = \mathrm{TRUE}$, set $A[i]$ to TRUE
4. If $A[n,B] = \mathrm{TRUE}$, accept $\langle B, x_1, \ldots, x_n \rangle$. Otherwise, reject

UNARY-SUBSET-SUM $\in$ P

Technique: dynamic programming

1. $A \leftarrow (n+1) \times (B+1)$ matrix.
2. Initialize $A[i, 0]$ to TRUE for all i; Initialize all other elements to FALSE
3. For $i = 1 \ldots n$:
   3.1 For $j = 1 \ldots B$:
      3.1.1 If $A[i-1, j] = \mathrm{TRUE}$, or if $j \geq x_i$ and $A[i-1, j-x_i] = \mathrm{TRUE}$, set $A[i]$ to TRUE
4. If $A[n, B] = \mathrm{TRUE}$, accept $\langle B, x_1, \ldots, x_n \rangle$. Otherwise, reject

- $O(n)$ outer loop iterations

UNARY-SUBSET-SUM $\in$ P

Technique: dynamic programming

1. $A \leftarrow (n+1) \times (B+1)$ matrix.
2. Initialize $A[i, 0]$ to TRUE for all i; Initialize all other elements to FALSE
3. For $i = 1 \ldots n$:
   - 3.1 For $j = 1 \ldots B$:
     - 3.1.1 If $A[i-1, j] = \text{TRUE}$, or if $j \geq x_i$ and $A[i-1, j-x_i] = \text{TRUE}$, set $A[i]$ to TRUE
4. If $A[n, B] = \text{TRUE}$, accept $\langle B, x_1, \ldots, x_n \rangle$. Otherwise, reject

- $O(n)$ outer loop iterations
- $O(B)$ inner loop iterations $= O(|\langle B \rangle|)$ since the input is <u>unary</u>

UNARY-SUBSET-SUM $\in$ P

Technique: dynamic programming

1. $A \leftarrow (n+1) \times (B+1)$ matrix.
2. Initialize $A[i, 0]$ to TRUE for all i; Initialize all other elements to FALSE
3. For $i = 1 \ldots n$:
   3.1 For $j = 1 \ldots B$:
      3.1.1 If $A[i-1, j] = \mathrm{TRUE}$, or if $j \geq x_i$ and $A[i-1, j-x_i] = \mathrm{TRUE}$, set $A[i]$ to TRUE
4. If $A[n, B] = \mathrm{TRUE}$, accept $\langle B, x_1, \ldots, x_n \rangle$. Otherwise, reject

- $O(n)$ outer loop iterations
- $O(B)$ inner loop iterations $= O(|\langle B \rangle|)$ since the input is unary
- $O(B \cdot n) \in \mathrm{P}$

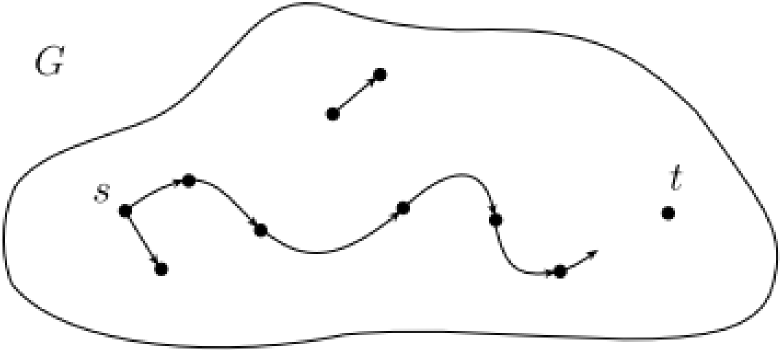
The language PATH

The language PATH

$\text{PATH} = \{\langle G, s, t\rangle | \text{G is a digraph with an s-t path}\}$

The language PATH

$$\text{PATH} = \{\langle G, s, t\rangle | \text{G is a digraph with an s-t path}\}$$



PATH $\in$ P

PATH $\in$ P

Technique: Perform a *breadth-first search*

PATH $\in$ P

Technique: Perform a *breadth-first search*

1. Mark node s

PATH $\in$ P

Technique: Perform a *breadth-first search*

1. Mark node s
2. Repeat the following until now additional nodes are marked

PATH $\in$ P

Technique: Perform a *breadth-first search*

1. Mark node s
2. Repeat the following until now additional nodes are marked
   2.1 Scan all edges. If there is an edge (u, v) where u is marked and v is unmarked, mark v

PATH $\in$ P

Technique: Perform a *breadth-first search*

1. Mark node s
2. Repeat the following until now additional nodes are marked
   2.1 Scan all edges. If there is an edge (u, v) where u is marked and v is unmarked, mark v
3. If t is marked, accept $\langle G, s, t \rangle$. Otherwise, reject.

PATH $\in$ P

Technique: Perform a *breadth-first search*

1. Mark node s
2. Repeat the following until now additional nodes are marked
   2.1 Scan all edges. If there is an edge (u, v) where u is marked and v is unmarked, mark v
3. If t is marked, accept $\langle G, s, t\rangle$. Otherwise, reject.

- $O(|V|)$ rounds

PATH $\in$ P

Technique: Perform a *breadth-first search*

1. Mark node s
2. Repeat the following until now additional nodes are marked
   2.1 Scan all edges. If there is an edge (u, v) where u is marked and v is unmarked, mark v
3. If t is marked, accept $\langle G, s, t\rangle$. Otherwise, reject.

- $O(|V|)$ rounds
- $O(|E|)$ edge lookups per round

$\mathrm{PATH} \in \mathrm{P}$

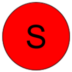
Technique: Perform a *breadth-first search*

1. Mark node s
2. Repeat the following until now additional nodes are marked
   2.1 Scan all edges. If there is an edge (u, v) where u is marked and v is unmarked, mark v
3. If t is marked, accept $\langle G, s, t \rangle$. Otherwise, reject.

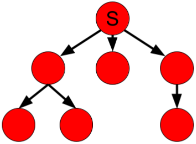
- $O(|V|)$ rounds
- $O(|E|)$ edge lookups per round
- $O(|V| \cdot |E|) \in \mathrm{P}$

PATH $\in$ P

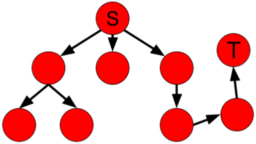
1. Mark vertex S



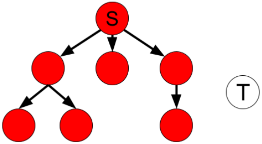
2. Mark all neighbors of S (and their neighbors, and so on)



3. Continue until T gets marked…



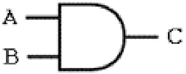
4. ...or until we can't mark further



Logical symbols

Logical symbols

AND

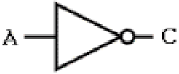


Inputs		Output
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

OR

Inputs		Output
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

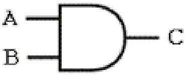
NOT



Input	Output
A	C
0	1
1	0

Logical symbols

AND

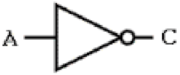


Inputs		Output
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

OR

Inputs		Output
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

NOT

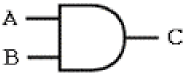


Input	Output
A	C
0	1
1	0

- AND ($\wedge$): all inputs must be TRUE

Logical symbols

AND

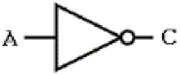


Inputs		Output
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

OR

Inputs		Output
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

NOT

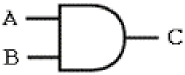


Input	Output
A	C
0	1
1	0

- AND ($\wedge$): all inputs must be TRUE
- OR ($\vee$): at least one input must be TRUE

Logical symbols

AND

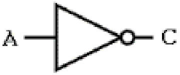


Inputs		Output
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

OR

Inputs		Output
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

NOT



Input	Output
A	C
0	1
1	0

- AND ($\wedge$): all inputs must be TRUE
- OR ($\vee$): at least one input must be TRUE
- NOT ($\neg$): input must be FALSE

Logical symbol practice

Suppose $x = \mathrm{TRUE}, y = \mathrm{TRUE}, z = \mathrm{FALSE}.$
Which of the following expressions are TRUE?

A) x

B) z

C) $y \vee z$

D) $\neg(x \wedge y)$

E) $(x \vee y) \wedge (y \vee z)$

F) $\neg x \vee (\neg y \vee \neg z)$

G) $(x \wedge y) \wedge (y \wedge z)$

H) $(x \vee y) \wedge (z \vee z \vee z)$

Logical symbol practice

Suppose $x = \text{TRUE}, y = \text{TRUE}, z = \text{FALSE}$.
Which of the following expressions are TRUE?

A) x ✓

B) z

C) $y \vee z$ ✓

D) $\neg(x \wedge y)$

E) $(x \vee y) \wedge (y \vee z)$ ✓

F) $\neg x \vee (\neg y \vee \neg z)$✓

G) $(x \wedge y) \wedge (y \wedge z)$

H) $(x \vee y) \wedge (z \vee z \vee z)$

Conjunctive Normal Form

Conjunctive Normal Form

Def: A **Conjunctive Normal Form (CNF) formula** is an expression of the following form:

Conjunctive Normal Form

Def: A **Conjunctive Normal Form (CNF) formula** is an expression of the following form:

1. Disjunction of several **clauses**

$$F = C_1 \wedge C_2 \wedge \ldots C_n$$

Conjunctive Normal Form

Def: A **Conjunctive Normal Form (CNF) formula** is an expression of the following form:

1. Disjunction of several **clauses**

$$F = C_1 \wedge C_2 \wedge \ldots C_n$$

2. Each clause is conjunction of several **variables**

$$C_i = (x_{i_1} \vee x_{i_2} \vee \ldots x_{i_n})$$

Conjunctive Normal Form

Def: A **Conjunctive Normal Form (CNF) formula** is an expression of the following form:

1. Disjunction of several **clauses**

$$F = C_1 \wedge C_2 \wedge \ldots C_n$$

2. Each clause is conjunction of several **variables**

$$C_i = (x_{i_1} \vee x_{i_2} \vee \ldots x_{i_n})$$

3. Each variable can be either positive x_i or negative $\neg x_i$

Conjunctive Normal Form

Def: A **Conjunctive Normal Form (CNF) formula** is an expression of the following form:

1. Disjunction of several **clauses**

$$F = C_1 \wedge C_2 \wedge \ldots C_n$$

2. Each clause is conjunction of several **variables**

$$C_i = (x_{i_1} \vee x_{i_2} \vee \ldots x_{i_n})$$

3. Each variable can be either positive x_i or negative $\neg x_i$

Examples:

Conjunctive Normal Form

Def: A **Conjunctive Normal Form (CNF) formula** is an expression of the following form:

1. Disjunction of several **clauses**

$$F = C_1 \wedge C_2 \wedge \ldots C_n$$

2. Each clause is conjunction of several **variables**

$$C_i = (x_{i_1} \vee x_{i_2} \vee \ldots x_{i_n})$$

3. Each variable can be either positive x_i or negative $\neg x_i$

Examples:

- $(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5)$

Conjunctive Normal Form

Def: A **Conjunctive Normal Form (CNF) formula** is an expression of the following form:

1. Disjunction of several **clauses**

$$F = C_1 \wedge C_2 \wedge \ldots C_n$$

2. Each clause is conjunction of several **variables**

$$C_i = (x_{i_1} \vee x_{i_2} \vee \ldots x_{i_n})$$

3. Each variable can be either positive x_i or negative $\neg x_i$

Examples:

- $(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5)$
- $(x_1 \vee \neg x_1) \wedge (x_2 \vee x_3 \vee x_4 \vee x_5 \vee \neg x_1) \wedge (\neg x_2)$

Conjunctive Normal Form

Which of the following expressions are in conjunctive normal form?

A) (x_1)
B) (x_2)
C) $(\neg x_1 \vee \neg x_1)$
D) $\neg(x_1 \vee x_1)$
E) $(x_1 \wedge x_2 \wedge x_3) \vee (x_4 \wedge x_5)$
F) $(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6)$
G) $(x_1 \vee x_2 \vee x_3) \vee (\neg x_1 \vee \neg x_2)$
H) $(x_1 \wedge x_2 \wedge x_3) \wedge (\neg x_1 \wedge \neg x_2)$

Conjunctive Normal Form

Which of the following expressions are in conjunctive normal form?

A) (x_1) ✓
B) (x_2) ✓
C) $(\neg x_1 \vee \neg x_1)$ ✓
D) $\neg(x_1 \vee x_1)$
E) $(x_1 \wedge x_2 \wedge x_3) \vee (x_4 \wedge x_5)$
F) $(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6)$ ✓
G) $(x_1 \vee x_2 \vee x_3) \vee (\neg x_1 \vee \neg x_2)$
H) $(x_1 \wedge x_2 \wedge x_3) \wedge (\neg x_1 \wedge \neg x_2)$

CNF Satisfying Assignment

CNF Satisfying Assignment

- **Def:** A **truth assignment** sets every variable to either TRUE or FALSE

CNF Satisfying Assignment

- **Def:** A **truth assignment** sets every variable to either TRUE or FALSE
  - **Note:** If x_i is FALSE then $\neg x_i$ is TRUE

CNF Satisfying Assignment

- **Def:** A **truth assignment** sets every variable to either TRUE or FALSE
  - **Note:** If x_i is FALSE then $\neg x_i$ is TRUE
- A CNF clause is **satisfied** if at least one of its variables is TRUE

CNF Satisfying Assignment

- **Def:** A **truth assignment** sets every variable to either TRUE or FALSE
  - **Note:** If x_i is FALSE then $\neg x_i$ is TRUE
- A CNF clause is **satisfied** if at least one of its variables is TRUE
- A CNF formula is satisfied if *all* of its clauses are satisfied

CNF Satisfying Assignment

- **Def:** A **truth assignment** sets every variable to either TRUE or FALSE
  - **Note:** If x_i is FALSE then $\neg x_i$ is TRUE
- A CNF clause is **satisfied** if at least one of its variables is TRUE
- A CNF formula is satisfied if *all* of its clauses are satisfied
- A CNF formula is **satisfiable** if there *exists* a satisfying assignment

CNF Satisfying Assignment

$$F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (x_2) \wedge (\neg x_5 \vee \neg x_1)$$

$x_1 = x_4 = x_5 = \text{TRUE}$
$x_2 = x_3 = \text{FALSE}$

Which clauses are satisfied?

A) $(x_1 \vee x_2 \vee x_3)$
B) $(\neg x_1 \vee x_3 \vee x_4)$
C) (x_2)
D) $(\neg x_5 \vee \neg x_1)$

CNF Satisfying Assignment

$$F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (x_2) \wedge (\neg x_5 \vee \neg x_1)$$

$x_1 = x_4 = x_5 = \text{TRUE}$
$x_2 = x_3 = \text{FALSE}$

Which clauses are satisfied?

A) $(x_1 \vee x_2 \vee x_3)$ ✓
B) $(\neg x_1 \vee x_3 \vee x_4)$ ✓
C) (x_2)
D) $(\neg x_5 \vee \neg x_1)$

CNF Satisfiability

$x_1 = x_4 = x_5 = \mathrm{TRUE}$
$x_2 = x_3 = \mathrm{FALSE}$

Which of the following formulas are satisfied?

A) $F = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_4 \vee x_5)$

B) $F = (x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4) \wedge (x_5)$

C) $F = (x_1) \wedge (x_2) \wedge (x_3) \wedge (x_4) \wedge (x_5)$

D) $F = (\neg x_1 \vee \neg x_4 \vee x_5) \wedge (x_2 \vee x_3)$

CNF Satisfiability

$x_1 = x_4 = x_5 = \text{TRUE}$
$x_2 = x_3 = \text{FALSE}$

Which of the following formulas are satisfied?

A) $F = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_4 \vee x_5)$ ✓

B) $F = (x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4) \wedge (x_5)$ ✓

C) $F = (x_1) \wedge (x_2) \wedge (x_3) \wedge (x_4) \wedge (x_5)$

D) $F = (\neg x_1 \vee \neg x_4 \vee x_5) \wedge (x_2 \vee x_3)$

CNF Satisfying Assignment

Which of the following formulas are satisfiable?

A) $F = (x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6)$

B) $F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$

C) $F = (x_1) \wedge (\neg x_2)$

D) $F = (x_1) \wedge (\neg x_1)$

CNF Satisfying Assignment

Which of the following formulas are satisfiable?

A) $F = (x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6)$ ✓

B) $F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$ ✓

C) $F = (x_1) \wedge (\neg x_2)$ ✓

D) $F = (x_1) \wedge (\neg x_1)$

CNF Satisiability

Is the following formula satisfiable?

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

CNF Satisiability

Is the following formula satisfiable?

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

These four clauses can't all be satisfied!

The language 2-SAT

The language 2-SAT

Def: A **2-CNF Formula** is a CNF formula with at most 2 variables in each clause

The language 2-SAT

Def: A **2-CNF Formula** is a CNF formula with at most 2 variables in each clause

$$2\text{-SAT} = \{F | F \text{ is a satisfiable 2-CNF formula}\}$$

The language 2-SAT

Def: A **2-CNF Formula** is a CNF formula with at most 2 variables in each clause

$$\text{2-SAT} = \{F \mid F \text{ is a satisfiable 2-CNF formula}\}$$

Which of these formulas are in the language 2-SAT?

A) $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$
B) $(x_1 \vee x_1) \wedge (\neg x_1 \vee \neg x_1)$
C) $(x_1) \wedge (x_2) \wedge (x_3)$
D) $(x_1 \vee x_2 \vee x_3)$

The language 2-SAT

Def: A **2-CNF Formula** is a CNF formula with at most 2 variables in each clause

$$\text{2-SAT} = \{F | F \text{ is a satisfiable 2-CNF formula}\}$$

Which of these formulas are in the language 2-SAT?

A) $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$ ✓
B) $(x_1 \vee x_1) \wedge (\neg x_1 \vee \neg x_1)$
C) $(x_1) \wedge (x_2) \wedge (x_3)$ ✓
D) $(x_1 \vee x_2 \vee x_3)$

Satisfying a 2-CNF Formula

Consider the following formula:

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

Satisfying a 2-CNF Formula

Consider the following formula:

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

- If x_1 is FALSE then x_2 must be FALSE

Satisfying a 2-CNF Formula

Consider the following formula:

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

- If x_1 is FALSE then x_2 must be FALSE
- If x_2 is FALSE, then x_3 must be TRUE

Satisfying a 2-CNF Formula

Consider the following formula:

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

- If x_1 is FALSE then x_2 must be FALSE
- If x_2 is FALSE, then x_3 must be TRUE
- If x_3 is TRUE then x_4 must be FALSE

Satisfying a 2-CNF Formula

Consider the following formula:

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

- If x_1 is FALSE then x_2 must be FALSE
- If x_2 is FALSE, then x_3 must be TRUE
- If x_3 is TRUE then x_4 must be FALSE
- If x_4 is FALSE then x_1 must be TRUE

Satisfy

Consi

$F = ($ … $\lor x_1)$

Satisfying a 2-CNF Formula

Consider the following formula:

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

- If x_1 is TRUE then x_4 must be TRUE

Satisfying a 2-CNF Formula

Consider the following formula:

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

- If x_1 is TRUE then x_4 must be TRUE
- If x_4 is TRUE, then x_3 must be FALSE

Satisfying a 2-CNF Formula

Consider the following formula:

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

- If x_1 is TRUE then x_4 must be TRUE
- If x_4 is TRUE, then x_3 must be FALSE
- If x_3 is FALSE then x_2 must be TRUE

Satisfying a 2-CNF Formula

Consider the following formula:

$$F = (x_1 \lor \neg x_2) \land (x_2 \lor x_3) \land (\neg x_3 \lor \neg x_4) \land (x_4 \lor \neg x_1)$$

- If x_1 is TRUE then x_4 must be TRUE
- If x_4 is TRUE, then x_3 must be FALSE
- If x_3 is FALSE then x_2 must be TRUE
- If x_2 is TRUE then x_1 must be TRUE - which it is!

2-SAT implication graph

2-SAT implication graph

- Suppose we have a clause $C = (x_i \vee x_j)$

2-SAT implication graph

- Suppose we have a clause $C = (x_i \lor x_j)$
- If x_i is FALSE then x_j must be TRUE

2-SAT implication graph

- Suppose we have a clause $C = (x_i \vee x_j)$
- If x_i is FALSE then x_j must be TRUE
  - $\neg x_i \implies x_j$

2-SAT implication graph

- Suppose we have a clause $C = (x_i \vee x_j)$
- If x_i is FALSE then x_j must be TRUE
  - $\neg x_i \implies x_j$
- If x_j is FALSE then x_i must be true

2-SAT implication graph

- Suppose we have a clause $C = (x_i \vee x_j)$
- If x_i is FALSE then x_j must be TRUE
  - $\neg x_i \implies x_j$
- If x_j is FALSE then x_i must be true
  - $\neg x_j \implies x_i$

2-SAT implication graph

- Suppose we have a clause $C = (x_i \vee x_j)$
- If x_i is FALSE then x_j must be TRUE
  - $\neg x_i \implies x_j$
- If x_j is FALSE then x_i must be true
  - $\neg x_j \implies x_i$
- If $x_i \implies x_j$ and $x_j \implies x_k$ then $x_i \implies x_k$ (transitive property)

2-SAT implication graph

- Suppose we have a clause $C = (x_i \vee x_j)$
- If x_i is FALSE then x_j must be TRUE
  - $\neg x_i \implies x_j$
- If x_j is FALSE then x_i must be true
  - $\neg x_j \implies x_i$
- If $x_i \implies x_j$ and $x_j \implies x_k$ then $x_i \implies x_k$ (transitive property)
- We can use an **implication graph** to represent these relationships

2-SAT implication graph

- Suppose we have a clause $C = (x_i \vee x_j)$
- If x_i is FALSE then x_j must be TRUE
  - $\neg x_i \implies x_j$
- If x_j is FALSE then x_i must be true
  - $\neg x_j \implies x_i$
- If $x_i \implies x_j$ and $x_j \implies x_k$ then $x_i \implies x_k$ (transitive property)
- We can use an **implication graph** to represent these relationships
  - Every node is variable

2-SAT implication graph

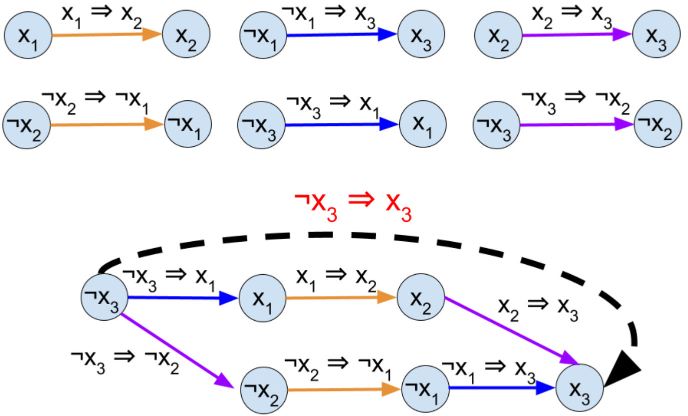
- Suppose we have a clause $C = (x_i \vee x_j)$
- If x_i is FALSE then x_j must be TRUE
  - $\neg x_i \implies x_j$
- If x_j is FALSE then x_i must be true
  - $\neg x_j \implies x_i$
- If $x_i \implies x_j$ and $x_j \implies x_k$ then $x_i \implies x_k$ (transitive property)
- We can use an **implication graph** to represent these relationships
  - Every node is variable
  - Every edge is an implication

2-SAT implication graph

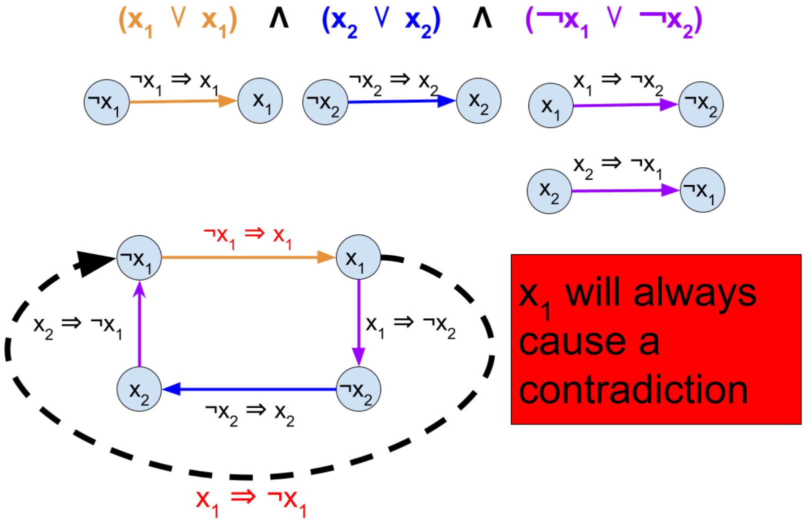
- Suppose we have a clause $C = (x_i \vee x_j)$
- If x_i is FALSE then x_j must be TRUE
  - $\neg x_i \implies x_j$
- If x_j is FALSE then x_i must be true
  - $\neg x_j \implies x_i$
- If $x_i \implies x_j$ and $x_j \implies x_k$ then $x_i \implies x_k$ (transitive property)
- We can use an **implication graph** to represent these relationships
  - Every node is variable
  - Every edge is an implication
  - **Every path is a (transitive) implication**

2-SAT implication graph

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



2-SAT implication graph



2-SAT $\in$ P

2-SAT ∈ P

Input: a formula F with n variables and m clauses

2-SAT $\in$ P

Input: a formula F with n variables and m clauses

1. Create the implication graph for F

2-SAT $\in$ P

Input: a formula F with n variables and m clauses

1. Create the implication graph for F
2. For every variable x_i do the following

2-SAT $\in$ P

Input: a formula F with n variables and m clauses

1. Create the implication graph for F
2. For every variable x_i do the following
   2.1 Check if there is a path from x_i to $\neg x_i$

2-SAT $\in$ P

Input: a formula F with n variables and m clauses

1. Create the implication graph for F
2. For every variable x_i do the following
   2.1 Check if there is a path from x_i to $\neg x_i$
   2.2 Check if there is a path from $\neg x_i$ to x_i

2-SAT $\in$ P

Input: a formula F with n variables and m clauses

1. Create the implication graph for F
2. For every variable x_i do the following
   2.1 Check if there is a path from x_i to $\neg x_i$
   2.2 Check if there is a path from $\neg x_i$ to x_i
   2.3 If both paths exist, there is a contradiction. Immediately reject F

2-SAT $\in$ P

Input: a formula F with n variables and m clauses

1. Create the implication graph for F
2. For every variable x_i do the following
   2.1 Check if there is a path from x_i to $\neg x_i$
   2.2 Check if there is a path from $\neg x_i$ to x_i
   2.3 If both paths exist, there is a contradiction. Immediately reject F
3. If there are no contradictions, accept F

2-SAT $\in$ P

Input: a formula F with n variables and m clauses

1. Create the implication graph for F
2. For every variable x_i do the following
   2.1 Check if there is a path from x_i to $\neg x_i$
   2.2 Check if there is a path from $\neg x_i$ to x_i
   2.3 If both paths exist, there is a contradiction. Immediately reject F
3. If there are no contradictions, accept F

- $O(n)$ vertices + $O(m)$ edges

2-SAT $\in$ P

Input: a formula F with n variables and m clauses

1. Create the implication graph for F
2. For every variable x_i do the following
   2.1 Check if there is a path from x_i to $\neg x_i$
   2.2 Check if there is a path from $\neg x_i$ to x_i
   2.3 If both paths exist, there is a contradiction. Immediately reject F
3. If there are no contradictions, accept F

- $O(n)$ vertices + $O(m)$ edges
- $O(n)$ loop iterations

2-SAT $\in$ P

Input: a formula F with n variables and m clauses

1. Create the implication graph for F
2. For every variable x_i do the following
   2.1 Check if there is a path from x_i to $\neg x_i$
   2.2 Check if there is a path from $\neg x_i$ to x_i
   2.3 If both paths exist, there is a contradiction. Immediately reject F
3. If there are no contradictions, accept F

- $O(n)$ vertices $+$ $O(m)$ edges
- $O(n)$ loop iterations
- PATH $\in$ P, each loop iteration is poly-time

2-SAT $\in$ P

Input: a formula F with n variables and m clauses

1. Create the implication graph for F
2. For every variable x_i do the following
   2.1 Check if there is a path from x_i to $\neg x_i$
   2.2 Check if there is a path from $\neg x_i$ to x_i
   2.3 If both paths exist, there is a contradiction. Immediately reject F
3. If there are no contradictions, accept F

- $O(n)$ vertices $+$ $O(m)$ edges
- $O(n)$ loop iterations
- $\mathrm{PATH} \in \mathrm{P}$, each loop iteration is poly-time
- $O(n) + O(m) + O(n) \cdot$ poly-time $\in \mathrm{P}$

The class EXP

The class EXP

- **Def:** The class EXP is the set of all languages that can be be decided in exponential time

The class EXP

- **Def:** The class EXP is the set of all languages that can be be decided in exponential time
  - $O(2^{n^c})$ for some constant c

The class EXP

- **Def:** The class EXP is the set of all languages that can be be decided in exponential time
  - $O(2^{n^c})$ for some constant c
- Alternate definition:

$$\mathrm{EXP} = \bigcup_{c} \mathrm{TIME}(T(2^{n^c}))$$

The class EXP

- **Def:** The class EXP is the set of all languages that can be be decided in exponential time
  - $O(2^{n^c})$ for some constant c
- Alternate definition:

$$\mathrm{EXP} = \bigcup_c \mathrm{TIME}(T(2^{n^c}))$$

- EXP languages are considered "intractable"

P vs. EXP

P vs. EXP

- **Note:** $\mathrm{P} \subseteq \mathrm{EXP}$

P vs. EXP

- **Note:** $\mathrm{P} \subseteq \mathrm{EXP}$
- Does $\mathrm{P} = \mathrm{EXP}$?

P vs. EXP

- **Note:** $\mathrm{P} \subseteq \mathrm{EXP}$
- Does $\mathrm{P} = \mathrm{EXP}$?
  - Can every exponential-time algorithm be converted to a polynomial-time algorithm?

Time hierarchy theorem

Time hierarchy theorem

- **Time Hierarchy Theorem:** $\mathrm{TIME}(T(n)) \subsetneq \mathrm{TIME}(T(2n)^3)$

Time hierarchy theorem

- **Time Hierarchy Theorem:** $\mathrm{TIME}(T(n)) \subsetneq \mathrm{TIME}(T(2n)^3)$
  - **Proof idea:** Use diagonalization to create a machine that contradicts all the $\mathrm{TIME}(T(n))$ machines

Time hierarchy theorem

- **Time Hierarchy Theorem:** $\mathrm{TIME}(T(n)) \subsetneq \mathrm{TIME}(T(2n)^3)$
  - **Proof idea:** Use diagonalization to create a machine that contradicts all the $\mathrm{TIME}(T(n))$ machines
  - The construction creates a machine that runs in time $O(T(2n)^3)$

Time hierarchy theorem

- **Time Hierarchy Theorem:** $\mathrm{TIME}(T(n)) \subsetneq \mathrm{TIME}(T(2n)^3)$
  - **Proof idea:** Use diagonalization to create a machine that contradicts all the $\mathrm{TIME}(T(n))$ machines
  - The construction creates a machine that runs in time $O(T(2n)^3)$
- **Glass half full**: More time will *always* allow us to solve more more problems

Time hierarchy theorem

- **Time Hierarchy Theorem:** $\mathrm{TIME}(T(n)) \subsetneq \mathrm{TIME}(T(2n)^3)$
  - **Proof idea:** Use diagonalization to create a machine that contradicts all the $\mathrm{TIME}(T(n))$ machines
  - The construction creates a machine that runs in time $O(T(2n)^3)$
- **Glass half full**: More time will *always* allow us to solve more more problems
- **Glass half empty**: Certain problems *can't* be solved within a certain amount of time

Time hierarchy theorem

Time hierarchy theorem

- **Time Hierarchy Theorem:** $\mathrm{TIME}(T(n)) \subsetneq \mathrm{TIME}(T(2n)^3)$
  - **Proof idea:** Use diagonalization to create a machine that contradicts all the $\mathrm{TIME}(T(n))$ machines
  - The construction creates a machine that runs in time $O(T(2n)^3)$
- **Glass half full**: More time will *always* allow us to solve more more problems
- **Glass half empty**: Certain problems *can't* be solved within a certain amount of time

$$\mathrm{P} \subseteq \mathrm{TIME}(2^n) \subsetneq \mathrm{TIME}((2^{2n})^3) \subseteq \mathrm{EXP}$$