CS 374A Final Review

big bad darth vader

ACM @ UIUC

May 4, 2025

Disclaimers and Logistics

- **Disclaimer:** Some of us are CAs, but we have not seen the exam. We have no idea what the questions are. However, we've taken the course and reviewed Jeffe's previous exams, so we have **suspicions** as to what the questions will be like.
- This review session is being recorded. Recordings and slides will be distributed on EdStem after the end.
- **Agenda:** We'll quickly review all topics likely to be covered, then go through a practice exam, then review individual topics by request.
  - Questions are designed to be written in the same style as Jeff's previous exams but to be *slightly* harder, so don't worry if you don't get everything right away!
- Please let us know if we're going too fast/slow, not speaking loud enough/speaking too loud, etc.
- If you have a question anytime during the review session, please ask! Someone else almost surely has a similar question.
- We'll provide a feedback form at the end of the session.

Table of Contents

Induction

Template

Let x be an *arbitrary* string/integer/etc.

Inductive Hypothesis: Assume for all k s.t. k is shorter/smaller/etc. than x that $P(k)$ (what we're trying to prove) holds.

Base Case: If $x = 0, \epsilon$, whatever your base case is, then ..., so $P(x)$ holds.

Inductive Step: If $x \neq 0, \epsilon$, whatever your base case is, then ..., so by the inductive hypothesis, ..., so $P(x)$ holds.

Thus, by the principle of induction, $P(x)$ holds.

Induction

Template

Let x be an *arbitrary* string/integer/etc.

Inductive Hypothesis: Assume for all k s.t. k is shorter/smaller/etc. than x that $P(k)$ (what we're trying to prove) holds.

Base Case: If $x = 0, \epsilon$, whatever your base case is, then ..., so $P(x)$ holds.

Inductive Step: If $x \neq 0, \epsilon$, whatever your base case is, then ..., so by the inductive hypothesis, ..., so $P(x)$ holds.

Thus, by the principle of induction, $P(x)$ holds.

Some tips:

- Always use strong induction.
- Write out your IH, base case, and inductive step out explicitly.
- Think about what you would like to know about your smaller/shorter numbers/strings.

Regular Languages/Expressions

- Built inductively on 3 operations:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$
  - $()$ are used to group expressions
  - (implicit) concatenation operator: $L(r_1 r_2) = \{xy : x \in L_1, y \in L_2\}$

Regular Languages/Expressions

- Built inductively on 3 operations:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$
  - $()$ are used to group expressions
  - (implicit) concatenation operator: $L(r_1 r_2) = \{xy : x \in L_1, y \in L_2\}$
- Closed under Union ($\cup$), intersection ($\cap$), concatenation ($\cdot$), kleene star (*), complement (^{C}), set difference ($\setminus$), and reverse (^{R})
  - ... but only finitely many applications of these operations

Regular Languages/Expressions

- Built inductively on 3 operations:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$
  - $()$ are used to group expressions
  - (implicit) concatenation operator: $L(r_1 r_2) = \{xy : x \in L_1, y \in L_2\}$
- Closed under Union ($\cup$), intersection ($\cap$), concatenation ($\cdot$), kleene star (*), complement (^{C}), set difference ($\setminus$), and reverse (^{R})
  - ... but only finitely many applications of these operations
- If trying to guess whether or not a language is regular, think about memory. DFAs only get finite memory!
  - You don't get to look back indefinitely.
  - If your language requires you to track a number or string indefinitely, it is not regular!

Regular Languages/Expressions

- Built inductively on 3 operations:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
  - $*$ is the Kleene star. $L(r_1^*) = L(r_1)^*$
  - $()$ are used to group expressions
  - (implicit) concatenation operator: $L(r_1 r_2) = \{xy : x \in L_1, y \in L_2\}$
- Closed under Union ($\cup$), intersection ($\cap$), concatenation ($\cdot$), kleene star (*), complement (^{C}), set difference ($\setminus$), and reverse (^{R})
  - ... but only finitely many applications of these operations
- If trying to guess whether or not a language is regular, think about memory. DFAs only get finite memory!
  - You don't get to look back indefinitely.
  - If your language requires you to track a number or string indefinitely, it is not regular!
- **Regex Design Tips:**
  - What strings are in your language? Which ones aren't? Note edge cases (specifically check ϵ).
  - Look for patterns and substrings that you definitely need to include or repeat.

DFAs

- DFA $M = (Q, A, \Sigma, s, \delta)$
  - Q - FINITE set of states
  - $A \subseteq Q$ - accepting states
  - Σ - input alphabet, usually $\{0, 1\}$
  - $s \in Q$ - start state
  - $\delta : Q \times \Sigma \to Q$ - transition function

DFAs

- DFA $M = (Q, A, \Sigma, s, \delta)$
  - Q - FINITE set of states
  - $A \subseteq Q$ - accepting states
  - Σ - input alphabet, usually $\{0, 1\}$
  - $s \in Q$ - start state
  - $\delta : Q \times \Sigma \rightarrow Q$ - transition function
- **Tips for Creating DFAs**:
  - Define your states exactly! What does it mean to be at each state?
  - Based on these definitions, when should you accept? Define A accordingly.
  - What state represents ϵ? Make that the start state. Make sure that if L accepts ϵ, you accept your start state.
  - How does reading in a 0 or 1 change each state? Define δ accordingly.

DFAs

- DFA $M = (Q, A, \Sigma, s, \delta)$
  - Q - FINITE set of states
  - $A \subseteq Q$ - accepting states
  - Σ - input alphabet, usually $\{0, 1\}$
  - $s \in Q$ - start state
  - $\delta : Q \times \Sigma \to Q$ - transition function
- **Tips for Creating DFAs**:
  - Define your states exactly! What does it mean to be at each state?
  - Based on these definitions, when should you accept? Define A accordingly.
  - What state represents ϵ? Make that the start state. Make sure that if L accepts ϵ, you accept your start state.
  - How does reading in a 0 or 1 change each state? Define δ accordingly.
- Notes:
  - To go from L to its complement, just switch the accepting and non-accepting states.
  - Every DFA is automatically an NFA.
  - Every regular language can be represented by a DFA. Every DFA represents a regular language.

Product Constructions

- Say I can build DFA M_1 keeping track of one property and DFA M_2 keeping track of another property. What if I want a DFA M that keeps track of both properties?
- You can combine the information of both DFAs into one product DFA.
- The accept states in your new DFA define whether I only accept strings with both properties or strings with one or the other or some other logical operation.

Product Constructions

- Say I can build DFA M_1 keeping track of one property and DFA M_2 keeping track of another property. What if I want a DFA M that keeps track of both properties?
- You can combine the information of both DFAs into one product DFA.
- The accept states in your new DFA define whether I only accept strings with both properties or strings with one or the other or some other logical operation.

Template

1. Define (drawing/formal) your first DFA $M_1 = (Q_1, A_1, \Sigma, s_1, \delta_1)$.
2. Define (drawing/formal) your second DFA $M_2 = (Q_2, A_2, \Sigma, s_2, \delta_2)$.
3. Define the product DFA $M = (Q, A, \Sigma, s, \delta)$ as follows:
   - $Q = Q_1 \times Q_2 = \{(q_1, q_2) | q_1 \in Q_1, q_2 \in Q_2\}$ - each state is a tuple
   - $s = (s_1, s_2)$ - just the tuple containing both start states
   - $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ - apply the first transition function to the first state and the second transition function to the second state
   - $A = \{(q_1, q_2) | q_1 \in A_1$ **and/or/etc.** $q_2 \in A_2\}$ - check if the first state is accepted by the first DFA, check if the second state is accepted by the second DFA, and accept based on the problem statement

NFAs

- NFA $N = (Q, A, \Sigma, s, \delta)$
  - Q - FINITE set of states
  - $A \subseteq Q$ - accepting states
  - Σ - input alphabet, usually $\{0, 1\}$
  - $s \subseteq Q$ - start state(s)
  - $\delta : Q \times (\Sigma \cup \epsilon) \rightarrow 2^Q$ - transition function

NFAs

- NFA $N = (Q, A, \Sigma, s, \delta)$
  - Q - FINITE set of states
  - $A \subseteq Q$ - accepting states
  - Σ - input alphabet, usually $\{0, 1\}$
  - $s \subseteq Q$ - start state(s)
  - $\delta : Q \times (\Sigma \cup \epsilon) \rightarrow 2^Q$ - transition function
- Changes
  - You can now have multiple start states!
  - We've added epsilon transitions.
  - We can transition to multiple different states.

NFAs

- NFA $N = (Q, A, \Sigma, s, \delta)$
  - Q - FINITE set of states
  - $A \subseteq Q$ - accepting states
  - Σ - input alphabet, usually $\{0, 1\}$
  - $s \subseteq Q$ - start state(s)
  - $\delta : Q \times (\Sigma \cup \epsilon) \rightarrow 2^Q$ - transition function
- Changes
  - You can now have multiple start states!
  - We've added epsilon transitions.
  - We can transition to multiple different states.
- **Tips for Creating NFAs**:
  - Make sure that your transition function consistently leads to a SET of states (whether that set is empty, has 1 state, or multiple states).
  - Use epsilon transitions to jump from one state to another without reading anything (usually when you want to be able to transition from one phase to another).

NFAs

- NFA $N = (Q, A, \Sigma, s, \delta)$
  - Q - FINITE set of states
  - $A \subseteq Q$ - accepting states
  - Σ - input alphabet, usually $\{0, 1\}$
  - $s \subseteq Q$ - start state(s)
  - $\delta : Q \times (\Sigma \cup \epsilon) \rightarrow 2^Q$ - transition function
- Changes
  - You can now have multiple start states!
  - We've added epsilon transitions.
  - We can transition to multiple different states.
- **Tips for Creating NFAs**:
  - Make sure that your transition function consistently leads to a SET of states (whether that set is empty, has 1 state, or multiple states).
  - Use epsilon transitions to jump from one state to another without reading anything (usually when you want to be able to transition from one phase to another).
- Notes
  - Every NFA can be converted to a DFA through power set construction. The DFA states are the power set of the NFA states.

Fooling Sets

- DFAs only care about which state you're in, and not how you got there
  - If two strings result in the same DFA state, any additional suffix added to both will also result in both strings being in the same state.

Fooling Sets

- DFAs only care about which state you're in, and not how you got there
  - If two strings result in the same DFA state, any additional suffix added to both will also result in both strings being in the same state.
  - Thus, if we have x, y, and we know that there exists a **distinguishing suffix** z s.t. $xz \in L, yz \notin L$, then x, y must be in *different* states for *any* DFA that accepts L

Fooling Sets

- DFAs only care about which state you're in, and not how you got there
  - If two strings result in the same DFA state, any additional suffix added to both will also result in both strings being in the same state.
  - Thus, if we have x, y, and we know that there exists a **distinguishing suffix** z s.t. $xz \in L, yz \notin L$, then x, y must be in *different* states for *any* DFA that accepts L
- A **fooling set** is a set of strings where there exists a distinguishing suffix between every pair of strings
- Myhill-Nerode: min DFA size = max fooling set size
  - Thus, languages with infinite fooling sets are *not* regular

Fooling Sets

- DFAs only care about which state you're in, and not how you got there
  - If two strings result in the same DFA state, any additional suffix added to both will also result in both strings being in the same state.
  - Thus, if we have x, y, and we know that there exists a **distinguishing suffix** z s.t. $xz \in L, yz \notin L$, then x, y must be in *different* states for *any* DFA that accepts L
- A **fooling set** is a set of strings where there exists a distinguishing suffix between every pair of strings
- Myhill-Nerode: min DFA size = max fooling set size
  - Thus, languages with infinite fooling sets are *not* regular
- **Tips and Tricks**
  - If L needs to keep track of a value with no bound, create a fooling set around the part you count up.
  - If you're using strings of the form $1^k, 0^p$, etc. when sampling elements of your fooling set a^i, a^j, you may assume WLOG that $i < j$.

Prove $\{0^n 1^n \mid n \geq 0\}$ is irregular.

Template adapted from *Fall 2025 Homework 3 solutions*.

Let $F = \{$ $\}$, which is infinite.

Let $x, y \in F$ with $x \neq y$. Thus, $x =$ and $y =$, where .

Let $z =$.

- Then, $xz \in L$ because .
- Then, $yz \notin L$ because .

Thus, z is a distinguishing suffix for x and y, so F is a fooling set for L.

Since F is infinite, L is not a regular language.

Note: We can also pick z so that $xz \notin L$ and $yz \in L$. Just do whichever is easier.

Language Transformations

- We have a language L we know is regular.
- We have a function f from strings to strings: $f(w) = x$.
- We define a transformed language in one of the following formats:
  1. $L' = \{f(w) | w \in L\}$
  2. $L' = \{w | f(w) \in L\}$
- We want to show that L' is regular by making a DFA/NFA that accepts L'.

Language Transformations

- We have a language L we know is regular.
- We have a function f from strings to strings: $f(w) = x$.
- We define a transformed language in one of the following formats:
  1. $L' = \{f(w) | w \in L\}$
  2. $L' = \{w | f(w) \in L\}$
- We want to show that L' is regular by making a DFA/NFA that accepts L'.

Template

- Since L is regular, there is a DFA $M = (Q, A, \Sigma, s, \delta)$ that accepts it.
- Now, we create a DFA/NFA $M' = (Q', A', \Sigma, s', \delta')$ that accepts L'
  1. $L' = \{f(w) | w \in L\}$ - M' reads in the results of $f(w) = x$, so we need to undo f to find w from x and then push w through the original DFA M to check acceptance
  2. $L' = \{w | f(w) \in L\}$ - M' reads in strings w, so we need to apply f to w to find $f(w) = x$ and then push x through the original DFA M to check acceptance
- Define the states Q' by thinking about what information you need to interpret the next letter you read - usually $Q \times \{\text{Information needed to convert } x \text{ to } w \text{ or } w \text{ to } x\}$
- Define the transition function δ so it passes w or x back to the original DFA - potentially using nondeterminism or epsilon transitions

Context-Free Languages/Grammars

- Formally, a context-free grammar is defined by
  - V - nonterminals/variables
  - T - terminals/symbols
  - $S \in V$ - start variable
  - P - set of production rules $A \rightarrow \alpha$ with $A \in V$ and $\alpha \in (V \cup T)^*$

Context-Free Languages/Grammars

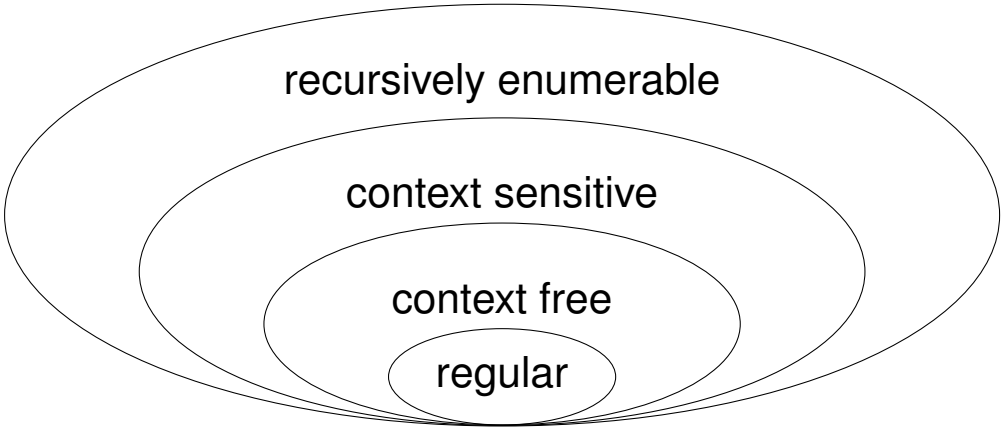
- Formally, a context-free grammar is defined by
  - V - nonterminals/variables
  - T - terminals/symbols
  - $S \in V$ - start variable
  - P - set of production rules $A \rightarrow \alpha$ with $A \in V$ and $\alpha \in (V \cup T)^*$
- Context-free grammars often build from the outside in, peeling away layer by layer

Context-Free Languages/Grammars

- Formally, a context-free grammar is defined by
  - V - nonterminals/variables
  - T - terminals/symbols
  - $S \in V$ - start variable
  - P - set of production rules $A \rightarrow \alpha$ with $A \in V$ and $\alpha \in (V \cup T)^*$
- Context-free grammars often build from the outside in, peeling away layer by layer
- CFLs are only closed under union, kleene star, and concatenation. CFLs are *not* closed under intersection or complement.

Context-Free Languages/Grammars

- Formally, a context-free grammar is defined by
  - V - nonterminals/variables
  - T - terminals/symbols
  - $S \in V$ - start variable
  - P - set of production rules $A \rightarrow \alpha$ with $A \in V$ and $\alpha \in (V \cup T)^*$
- Context-free grammars often build from the outside in, peeling away layer by layer
- CFLs are only closed under union, kleene star, and concatenation. CFLs are *not* closed under intersection or complement.



Practice Questions I

1. Let L be a regular language. Then, $L \cap \{0^n 1^n : n > 0\}$ is...
   (a) always regular
   (b) always irregular, but always context-free
   (c) sometimes irregular, but always context-free
   (d) sometimes non-context-free

Practice Questions I

1. Let L be a regular language. Then, $L \cap \{0^n 1^n : n > 0\}$ is...
   (a) always regular
   (b) always irregular, but always context-free
   (c) sometimes irregular, but always context-free
   (d) sometimes non-context-free
2. Define $\textsc{coverEvens}(w_1 w_2 \cdots w_n) = k_1 k_2 \cdots k_n$ and $\textsc{coverExponential}(w_1 w_2 \cdots w_n) = c_1 c_2 \cdots c_n$, where $k_i = \begin{cases} 1 & \text{if } i \equiv 0 \pmod 2 \\ w_i & \text{otherwise} \end{cases}$ and $c_i = \begin{cases} 1 & \text{if } \exists n \in \mathbf{Z} \text{ s.t. } i = 2^n \\ w_i & \text{otherwise} \end{cases}$.
   Which of the following is true?
   (a) If L is a regular language, then $\textsc{coverEvens}(L)$ is regular.
   (b) If $\textsc{coverEvens}(L)$ is a regular language, then L is regular.
   (c) If L is a regular language, then $\textsc{coverExponential}(L)$ is regular.
   (d) If $\textsc{coverExponential}(L)$ is a regular language, then L is regular.
   (e) Exactly two of the above are true
   (f) Exactly four of the above are true

Practice Questions I

1. Let L be a regular language. Then, $L \cap \{0^n 1^n : n > 0\}$ is...
   (a) always regular
   (b) always irregular, but always context-free
   (c) sometimes irregular, but always context-free
   (d) sometimes non-context-free
2. Define $\textsc{coverEvens}(w_1 w_2 \cdots w_n) = k_1 k_2 \cdots k_n$ and $\textsc{coverExponential}(w_1 w_2 \cdots w_n) = c_1 c_2 \cdots c_n$, where $k_i = \begin{cases} 1 & \text{if } i \equiv 0 \pmod 2 \\ w_i & \text{otherwise} \end{cases}$ and $c_i = \begin{cases} 1 & \text{if } \exists n \in \mathbf{Z} \text{ s.t. } i = 2^n \\ w_i & \text{otherwise} \end{cases}$.
   Which of the following is true?
   (a) If L is a regular language, then $\textsc{coverEvens}(L)$ is regular.
   (b) If $\textsc{coverEvens}(L)$ is a regular language, then L is regular.
   (c) If L is a regular language, then $\textsc{coverExponential}(L)$ is regular.
   (d) If $\textsc{coverExponential}(L)$ is a regular language, then L is regular.
   (e) Exactly two of the above are true
   (f) Exactly four of the above are true
3. If we instead define $\textsc{UnCoverEvens}(L)$ to be $\{w : \textsc{CoverEvens}(w) \in L\}$, then would $\textsc{UnCoverEvens}(L)$ be regular for all regular L?
   (a) Yes
   (b) No

Practice Questions II

Practice Questions II

4. Consider the language L consisting of the binary representation of all numbers congruent to 173 mod 374.
   (a) L does not have a fooling set.
   (b) L has a fooling set of size 173.
   (c) L has a fooling set of size 374.
   (d) L has a fooling set of size 375.
   (e) L has an infinite fooling set.

Practice Questions II

4. Consider the language L consisting of the binary representation of all numbers congruent to 173 mod 374.
   (a) L does not have a fooling set.
   (b) L has a fooling set of size 173.
   (c) L has a fooling set of size 374.
   (d) L has a fooling set of size 375.
   (e) L has an infinite fooling set.
5. Given a DFA M with n states, the minimum length of a string that M must accept (if $L(M)$ is non-empty) is at most:
   (a) n
   (b) $n - 1$
   (c) 2^n
   (d) $2^n - 1$
   (e) n^2

Practice Questions III

6. If L_1 is regular and L_2 is undecidable, then $L_1 \cap L_2$ is:
   (a) Always regular
   (b) Always context free
   (c) Always undecidable
   (d) Always decidable
   (e) Could be decidable or undecidable depending on L_1

Practice Questions III

6. If L_1 is regular and L_2 is undecidable, then $L_1 \cap L_2$ is:
   (a) Always regular
   (b) Always context free
   (c) Always undecidable
   (d) Always decidable
   (e) Could be decidable or undecidable depending on L_1
7. Given a language L, which of these is NOT necessarily true if L is context-free?
   (a) L^* is context-free
   (b) $L \cup \{\epsilon\}$ is context-free
   (c) $L \cap R$ is context-free for any regular language R
   (d) L^R (reverse of L) is context-free
   (e) $\{w\#w \mid w \in L\}$ is context-free

Practice Questions IV

1. Find a regexes for the following languages:
   (a) $\{0^a b^b 0^c \mid a \geq 0 \text{ and } b \geq 0 \text{ and } c \geq 0 \text{ and } a \equiv b + c \pmod 2\}$

Practice Questions IV

1. Find a regexes for the following languages:
   (a) $\{0^a b^b 0^c \mid a \geq 0 \text{ and } b \geq 0 \text{ and } c \geq 0 \text{ and } a \equiv b + c \pmod 2\}$
   (b) All strings that contain the substring `01` an odd number of times.

Practice Questions IV

1. Find a regexes for the following languages:
   (a) $\{0^a b^b 0^c \mid a \geq 0 \text{ and } b \geq 0 \text{ and } c \geq 0 \text{ and } a \equiv b + c \pmod 2\}$
   (b) All strings that contain the substring `01` an odd number of times.
2. Formally define DFAs/NFAs that accepts the following languages:
   (a) All strings whose ninth-to-last symbol is `0`, or equivalently, the set

$$\{x \circ z \mid x \in \Sigma^* \text{ and } z \in \Sigma^8\}.$$

Practice Questions IV

1. Find a regexes for the following languages:
   (a) $\{0^a b^b 0^c \mid a \geq 0 \text{ and } b \geq 0 \text{ and } c \geq 0 \text{ and } a \equiv b + c \pmod 2\}$
   (b) All strings that contain the substring `01` an odd number of times.
2. Formally define DFAs/NFAs that accepts the following languages:
   (a) All strings whose ninth-to-last symbol is `0`, or equivalently, the set

$$\{x \circ z \mid x \in \Sigma^* \text{ and } z \in \Sigma^8\}.$$

   (b) All strings w such that

$$(\#(0, w) \bmod 3) + (\#(1, w) \bmod 7) = (|w| \bmod 4).$$

Recursion

- **Definition:** Reducing the problem to a smaller instance of itself, where eventually we can terminate in a base case.
  - Think: If we have a problem of size n, we want to continuously reduce to a problem smaller than n.
  - Example: Binary search, Karatsuba

Template

```
1: procedure AmazingRecursiveAlgo(n)
2:     if n == [some base case] then
3:         return [value]
4:     else
5:         return AmazingRecursiveAlgo(n/2)
```

- Similar to **induction**!

Recursion: Runtime Analysis

- **General Form:**

$$T(n) = \underbrace{r}_{\text{\# of subproblems}} \cdot \overbrace{T\left(\frac{n}{c}\right)}^{\text{work at each subproblem}} + \underbrace{f(n)}_{\text{work at current level}}$$

  - Describes how the amount of work changes between each level of recursion.
  - We can solve for a **time complexity** that describes the scaling behavior of the algorithm at hand.

Recursion: Runtime Analysis

- **General Form:**

$$T(n) = \underbrace{r}_{\text{\# of subproblems}} \cdot \overbrace{T\left(\frac{n}{c}\right)}^{\text{work at each subproblem}} + \underbrace{f(n)}_{\text{work at current level}}$$

  - Describes how the amount of work changes between each level of recursion.
  - We can solve for a **time complexity** that describes the scaling behavior of the algorithm at hand.
- **Master Theorem**

Master Theorem

$$\text{Decreasing: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa < 1 \implies T(n) = O(f(n))$$
$$\text{Equal: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa = 1 \implies T(n) = O(f(n) \cdot log_c n)$$
$$\text{Increasing: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa > 1 \implies T(n) = O(n^{\log_c r})$$

Recursion: Runtime Analysis

- **General Form:**

$$T(n) = \underbrace{r}_{\text{\# of subproblems}} \cdot \overbrace{T\left(\frac{n}{c}\right)}^{\text{work at each subproblem}} + \underbrace{f(n)}_{\text{work at current level}}$$

  - Describes how the amount of work changes between each level of recursion.
  - We can solve for a **time complexity** that describes the scaling behavior of the algorithm at hand.
- **Master Theorem**

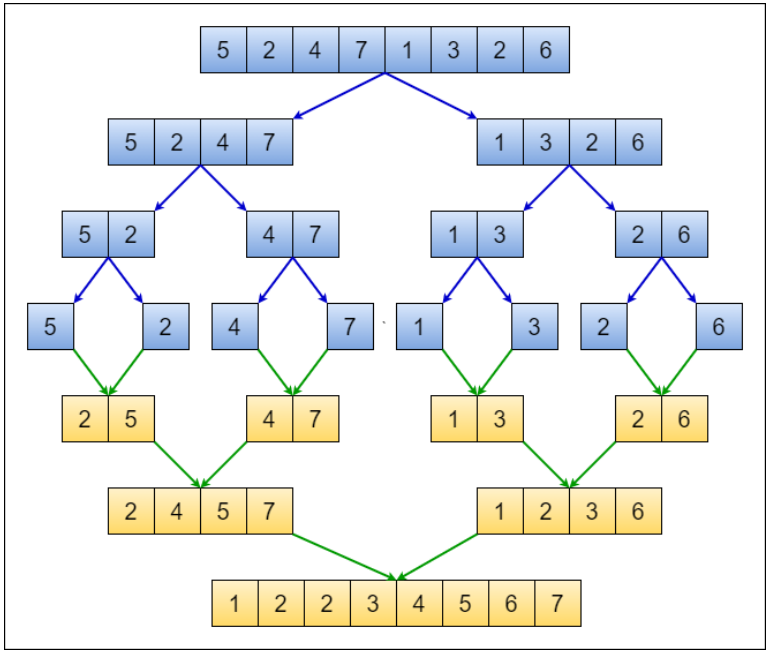
Master Theorem

$$\text{Decreasing: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa < 1 \implies T(n) = O(f(n))$$
$$\text{Equal: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa = 1 \implies T(n) = O(f(n) \cdot log_c n)$$
$$\text{Increasing: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa > 1 \implies T(n) = O(n^{\log_c r})$$

  - **Intuition:** If each level contains more work than the level below it, then the root level will dominate. If each level contains the same amount of work, then we have $log_c n$ levels with $f(n)$ work. If each level contains less work than the work below it, then the leaf nodes will dominate.

Divide and Conquer Algos: Merge Sort

- **Purpose:** Sort an arbitrary array.
- **Time Complexity**: $O(n \log n)$
- **Intuition:** Three phases: (a) split the array in half, (b) sort each side, (c) merge the sorted halves by repeatedly comparing smallest elements on each side not yet inserted.

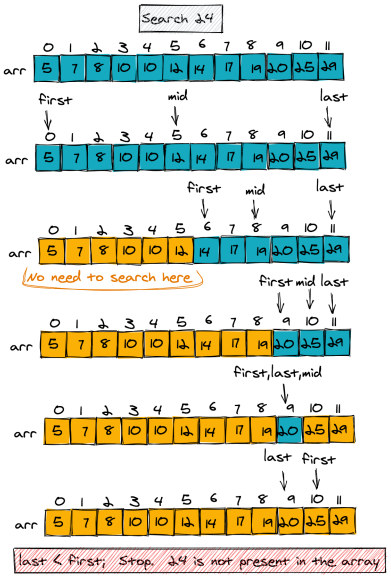


Divide and Conquer Algos: MomSelect

- **Purpose:** Get the r^{th} smallest element in an arbitrary array.
- **Time Complexity**: $O(n)$
- **Intuition**: Pick a pivot P with a value P_V and rearrange the array such that all the elements that are less than P_V are to the left of P and all the elements that are greater than P_V are to the right of P. If the length of the array of elements that are less than P_V is greater than r, then we know that the r^{th} smallest element is to the left of P and we recurse on the left subarray. Otherwise, we know that the r^{th} smallest element is to the right of P and we recurse on the right subarray.

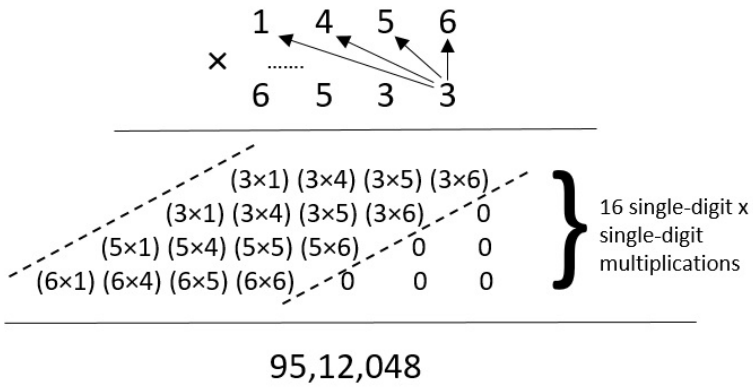
Divide and Conquer Algos: Binary Search

- **Purpose:** Find the existence of an element in a sorted array
- **Time Complexity**: $O(\log n)$
- **Intuition**: Say we are trying to find the value k. Pick the middle element M in the array. If $k > M$, the element must be to the right of M and we recurse on the right. Otherwise, we recurse on the left.



Divide and Conquer Algos: Karatsuba's

- **Purpose:** Multiplication
- **Time Complexity**: $O(n^{\log_2 3}) \approx O(n^{1.585})$
- 3 Phases:
  - **Divide**: Represent x as $x_0 p^n + x_1$, y as $y_0 p^n + y_1$
  - **Recurse**: Calculate $x_0 y_0$, $x_1 y_1$, $(x_0 + x_1)(y_0 + y_1)$
  - **Combine**: Use the three results to calculate our final answer.



Dynamic Programming

- It's backtracking, but we compute all of the subproblems iteratively.
  - This idea of "writing things down" as to not repeat computation is called **memoization**

Dynamic Programming

- It's backtracking, but we compute all of the subproblems iteratively.
  - This idea of "writing things down" as to not repeat computation is called **memoization**
- Alternatively, you can think about this recursively, except we check our memoization structure to see if we've computed anything before. If we have, we just use the computed result. Otherwise, we compute the subproblem.

Dynamic Programming

- It's backtracking, but we compute all of the subproblems iteratively.
  - This idea of "writing things down" as to not repeat computation is called **memoization**
- Alternatively, you can think about this recursively, except we check our memoization structure to see if we've computed anything before. If we have, we just use the computed result. Otherwise, we compute the subproblem.
- For a DP solution, we need:
  1. English Description
  2. Recurrence
  3. Memoization Structure
  4. Solution Location
  5. Evaluation Order
  6. Runtime

Dynamic Programming

- It's backtracking, but we compute all of the subproblems iteratively.
  - This idea of "writing things down" as to not repeat computation is called **memoization**
- Alternatively, you can think about this recursively, except we check our memoization structure to see if we've computed anything before. If we have, we just use the computed result. Otherwise, we compute the subproblem.
- For a DP solution, we need:
  1. English Description
  2. Recurrence
  3. Memoization Structure
  4. Solution Location
  5. Evaluation Order
  6. Runtime
- **How to solve a DP:**
  - Identify how we can take advantage of a recursive call on a smaller subset of the input space.
  - Identity base cases
  - Identity recurrences (they should cover all possible cases at each step)

Dynamic Programming: Longest Increasing Subsequence

1. **English Description:** $\text{LIS}(i, j)$ = the length of the longest increasing subsequence in $A[i..n]$, such that every element in the subsequence is $> A[j]$. Here, $j = 0$ can be used as a sentinel with $A[0] = -\infty$.
2. **Recurrence:**

$$\text{LIS}(i, j) = \begin{cases} 0 & \text{if } i > n, \\ \text{LIS}(i+1, j) & \text{if } A[i] \leq A[j], \\ \max\left(\text{LIS}(i+1, j),\ 1 + \text{LIS}(i+1, i)\right) & \text{if } A[i] > A[j]. \end{cases}$$

3. **Memoization Structure:** A 2D array $LIS[0..n, 0..n]$, storing all subproblem solutions $\text{LIS}(i, j)$.
4. **Solution Location:** The length of the LIS of the entire array is found at $\text{LIS}(1, 0)$, starting from the first element with the sentinel.
5. **Evaluation Order:** Fill the table in decreasing order of i.
6. **Runtime:** The table has $O(n^2)$ entries, each computed in $O(1)$, so the total runtime is $O(n^2)$.

Graphs

- **Definition:** A set of vertices V connected by a set of edges E. Individual edges are notated as (u, v), where $u, v \in V$.
  - They are usually represented as **adjacency lists** or **adjacency matrices**

Graphs

- **Definition:** A set of vertices V connected by a set of edges E. Individual edges are notated as (u, v), where $u, v \in V$.
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge $(u, v) \in E$ now has a direction $u \rightarrow v$

Graphs

- **Definition:** A set of vertices V connected by a set of edges E. Individual edges are notated as (u, v), where $u, v \in V$.
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge $(u, v) \in E$ now has a direction $u \rightarrow v$
  - **Acyclic:** No cycles.

Graphs

- **Definition:** A set of vertices V connected by a set of edges E. Individual edges are notated as (u, v), where $u, v \in V$.
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge $(u, v) \in E$ now has a direction $u \rightarrow v$
  - **Acyclic:** No cycles.
- **Path:** A sequence of distinct vertices where each pair of consecutive vertices have an edge

Graphs

- **Definition:** A set of vertices V connected by a set of edges E. Individual edges are notated as (u, v), where $u, v \in V$.
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge $(u, v) \in E$ now has a direction $u \rightarrow v$
  - **Acyclic:** No cycles.
- **Path:** A sequence of distinct vertices where each pair of consecutive vertices have an edge
- **Cycle:** A sequence of distinct vertices where each pair of consecutive vertices have an edge **and** the first and last vertices are connected.

Graph Algorithms: Traversal

Graph Algorithms: Traversal

- **BFS:**
  - **Purpose:** Reachability, Shortest Path (unweighted graph)
  - **Implementation details:** Add your neighbours to a **queue**, pop from the queue to get next node
  - **Runtime:** $O(V + E)$

Graph Algorithms: Traversal

- **BFS:**
  - **Purpose:** Reachability, Shortest Path (unweighted graph)
  - **Implementation details:** Add your neighbours to a **queue**, pop from the queue to get next node
  - **Runtime:** $O(V + E)$
- **DFS:**
  - **Purpose:** Reachability, toposort
  - **Implementation details:** Add your neighbours to a **stack**, pop from the stack to get next node
  - **Runtime:** $O(V + E)$

Graph Algorithms: Shortest Path

- **Dijkstra's**
  - **Purpose:** SSSP, no negative edges
  - **Implementation:** Visit neighbours in **priority queue**
  - **Runtime:** $O(m \log n)$ (with **Fibonacci Heap**, $O(m + n \log n)$)

Graph Algorithms: Shortest Path

- **Dijkstra's**
  - **Purpose:** SSSP, no negative edges
  - **Implementation:** Visit neighbours in **priority queue**
  - **Runtime:** $O(m \log n)$ (with **Fibonacci Heap**, $O(m + n \log n)$)
- **Bellman-Ford:**
  - **Purpose:** SSSP, yes negative weights. Will detect negative cycles.
  - **Implementation:** Dynamic Programming recurrence:
    - $d(v, k)$ is the shortest-walk distance from s to v using at most k edges
    - $d(v, k) = \min\left(d(v, k-1), \min_{u \to v} d(u, k-1) + \ell(u \to v)\right)$
  - **Runtime:** $O(mn)$

Graph Algorithms: Longest Path

Longest path in a Directed Acyclic Graph (DAG)

- **Purpose:** To find the longest simple path (no repeating vertices) by weight in a graph which is guaranteed to be a DAG[1].
- **Runtime[2]:** $O(V + E)$
- **Returns:** The sum of the weights of the longest path in the DAG.

[1] Finding the longest path in other types of graphs is at least NP-hard.
[2] This is a relatively straight-forward DP on a DAG problem if you wish to derive it.

Graph Problems: General Stuff

How to solve graph problems:

1. Identify type of problem (Reachability, Shortest Path, SCC)
2. Construct new graph
   - Add sources/sinks
   - Add vertices via $V' = V \times \{\text{some set}\}$ (Useful for tracking states)
   - Add vertices via $E' = E \times \{\text{some set}\}$ (Useful for allowing/prohibit certain behaviour)
3. Apply some stock algorithm **(DO NOT MODIFY THE ALGORITHMS - MODIFY THE INPUTS!)**
4. Draw connection between how to result of the algorithm upon the new graph relates to the solution of the original question.

Graphs?

The adjacency matrix A of a graph G has some nice properties that are useful in applications. One such property is that $A^\ell[i, j]$ is the number of walks using ℓ edges between vertices i and j. Describe and analyze an efficient algorithm to compute A^ℓ for a graph G with n vertices. You may use the fact that two $n \times n$ matrices can be multiplied in $O(n^3)$ time.

Dancing With the Stars

Sarah loves *Dancing With the Stars*. She is very particular about the performances she highlights in the finale montage: she wants a sequence of routines that maximizes excitement, but if she pays attention to one routine, she won't be able to pay attention to the next two. Can we pick k routines for her to watch that maximizes her excitement?

There are n weekly routines, indexed 1 through n. Each routine i has an excitement score $E[i]$.
Sarah can choose to watch a routine (add $E[i]$ to the total excitement) or not pay attention to it. If she chooses to watch a routine, she can't watch the next two routines. She can choose at most k routines.

Compute a subset of of $\leq k$ routines to watch that **maximizes total excitement**.

Secret Lives of Mormon Wives

Navid's favorite show is *The Secret Lives of Mormon Wives*, which he claims to watch "for the plot" but really watches for the unhinged drama. The show revolves around MomTok, questionable business ventures, and an alarming number of MLM schemes. As a result, there are many explosive secrets (affairs, business betrayals, Instagram unfollowings), and some secrets are eventually revealed to the audience through tearful confessionals and passive-aggressive Instagram stories. The timing of these revelations affects how much total drama the audience experiences. You are in charge of choosing which secrets to reveal.

Each secret is assigned to a unique episode, and a secret can only be revealed in its assigned episode.

There are n secrets, indexed 1 through n in episode order. Each secret i has a drama value $V[i]$ if it is revealed.

If two revealed secrets $i < j$ are revealed *consecutively* among the chosen secrets and occur $k = j - i$ episodes apart, they produce a bonus drama of $k - 1$ since this increases suspense.

Suppose $n = 6$ with drama values $V = [20,\ 1,\ 30,\ 2,\ 5,\ 14]$, and we choose to reveal secrets $\{1,\ 3,\ 6\}$. Total drama: $20 + (30 + (3 - 1 - 1)) + (14 + (6 - 3 - 1)) = 67$.

Design and analyze an algorithm to compute a subset of secrets to reveal that **maximizes the total drama**.

Bad Apple

Jeff is falling into a Bad Apple rabbit hole on Youtube. He wants to get from the original **Bad Apple** video to the video **bad apple but it's a sorting algorithm**[3] by *only* watching videos that are recommended to him in sequence.

Formally, you are given a list of n videos V such that each $v \in V$ has a watch time $\tau(v)$ and list $R(v) \subseteq V$ of sidebar recommended videos that can be watched after v is watched. Suppose video $s \in V$ is the original **Bad Apple** video and $t \in V$ is the **bad apple but it's a sorting algorithm** video. We call a sequence of videos starting from s and ending at t a *Terpsichorean marathon*.

(a) Describe and analyze an algorithm to find the *Terpsichorean marathon* of minimum total watch time.

(b) We call a video a *banger* if it is in some *Terpsichorean marathon* with minimum watch time. Design and analyze an efficient algorithm to find all the *banger* videos.

[3] Yes, this is a real thing.

Who's the number one spice?

Young Cardamom is bringing the flavor to the fish![4] His dish needs to have both **HEAT** and **SWEET** spices to be complete.

The kitchen can be modeled as a DAG where:

- Vertices represent prep stations
- Each station is labeled either HEAT, SWEET, or NEUTRAL
- A directed edge (u, v) with weight $f(u, v)$ means station u can send ingredients to station v, contributing flavor value $f(u, v)$
- Young Cardamom starts at station s and must reach plating station t

To achieve the #1 spice, Young Cardamom's path must visit **at least one HEAT station** and **at least one SWEET station**.

Design and analyze an algorithm to find a path from s to t that visits both required station types and maximizes the total flavor value.

[4] Of course, HAB is bringing the flavor to the rice.

TEAM UPDATE:

The 374 exam is being coordinated in a Signal group chat, and Pranay accidentally adds a student to the group chat. This leaks the exam to the student! Oops. You want to find out how quickly the exam can reach you.

We model the students' friend network as an unweighted, undirected[5] graph $G = (V, E)$, where each vertex is a student in the class and an edge $uv \in E$ means u and v are friends. Some students $D \subseteq V$ are in an illicit Discord server and can send the exam to each other even if there are no edges between them. However, Discord is quite a slow platform, so sending the message between two vertices in D takes $d > 1$ time. Otherwise, sending the exam between friends takes time 1.

Design and analyze an algorithm to determine the shortest amount of time needed for the exam to travel from the student added to the Signal group, s, to you, t.

[5] Unlike real life, friendships in this world are always bi-directional.

P and NP

- A **decision problem** is a problem with a true/false answer. (yes/no, etc.)
- **P** is the set of decision problems with a polynomial-time solver.
- **NP** is the set of decision problems with a polynomial-time *nondeterministic* solver.
- Alternatively, NP is the set of decision problems with a polynomial-time *certifier* for "true" answers, given a polynomial-size *certificate*.
  - Intuitively, with an NP problem, we can verify a "yes" answer quickly if we have the solution in front of us.
- For example, consider the yes/no problem of deciding whether a graph $G = (V, E)$ has a path containing all its vertices. (Hamiltonian Path)
  - If you were given the path already ($O(V)$ length) as a certificate, you could certify that the answer is "yes" in polynomial time.
  - Therefore, this problem is in NP.

P and NP

- A **decision problem** is a problem with a true/false answer. (yes/no, etc.)
- **P** is the set of decision problems with a polynomial-time solver.
- **NP** is the set of decision problems with a polynomial-time *nondeterministic* solver.
- Alternatively, NP is the set of decision problems with a polynomial-time *certifier* for "true" answers, given a polynomial-size *certificate*.
  - Intuitively, with an NP problem, we can verify a "yes" answer quickly if we have the solution in front of us.
- For example, consider the yes/no problem of deciding whether a graph $G = (V, E)$ has a path containing all its vertices. (Hamiltonian Path)
  - If you were given the path already ($O(V)$ length) as a certificate, you could certify that the answer is "yes" in polynomial time.
  - Therefore, this problem is in NP.

P and NP

- A **decision problem** is a problem with a true/false answer. (yes/no, etc.)
- **P** is the set of decision problems with a polynomial-time solver.
- **NP** is the set of decision problems with a polynomial-time *nondeterministic* solver.
- Alternatively, NP is the set of decision problems with a polynomial-time *certifier* for "true" answers, given a polynomial-size *certificate*.
  - Intuitively, with an NP problem, we can verify a "yes" answer quickly if we have the solution in front of us.
- For example, consider the yes/no problem of deciding whether a graph $G = (V, E)$ has a path containing all its vertices. (Hamiltonian Path)
  - If you were given the path already ($O(V)$ length) as a certificate, you could certify that the answer is "yes" in polynomial time.
  - Therefore, this problem is in NP.

co-NP

- **co-NP** is the set of decision problems X whose complements $\overline{X}$ are in NP.
- Alternatively, NP is the set of decision problems with a polynomial-time certifier for **"false"** answers, given a polynomial-size certificate.
- For example, the problem of deciding whether a graph *doesn't* have a Hamiltonian path is in co-NP.
- co-NP isn't on your skillset, but be aware that this is *not* the same thing as NP.

NP-Hard and NP-Complete

- **NP-Hard** is the set of decision problems that are at least as hard as NP...
  - Say you had an NP-Hard problem A.
  - If you could solve A in polynomial time, you could also solve *any* NP problem in polynomial time.
- We'll prove that problems are NP-hard by providing **polynomial-time** reductions from a known NP-hard problem to the problem in question.
- A problem is **NP-complete** if it is both in NP and NP-hard, that is, it is exactly as hard as NP.

Reductions I: Intuition

- **Intuition:** Problem B is "at least as hard" as problem A if we can use a black-box problem B solver (B oracle) to solve problem A with limited overhead (generally, polynomial-time).

Reductions I: Intuition

- **Intuition:** Problem B is "at least as hard" as problem A if we can use a black-box problem B solver (B oracle) to solve problem A with limited overhead (generally, polynomial-time).
- We know a variety of "hard" problems, so if we want to show that a problem B is hard, we need to show that oracles can be used to quickly solve some hard problem A (even if we believe that that oracle doesn't exist!).

Reductions I: Intuition

- **Intuition:** Problem B is "at least as hard" as problem A if we can use a black-box problem B solver (B oracle) to solve problem A with limited overhead (generally, polynomial-time).
- We know a variety of "hard" problems, so if we want to show that a problem B is hard, we need to show that oracles can be used to quickly solve some hard problem A (even if we believe that that oracle doesn't exist!).
- This is building a **reduction from** problem A **to** problem B!
- Essentially, if I want to show that Problem B is hard (and I already know that Problem A is hard), I need to assume that Problem B is easy and then solve Problem A using Problem B.
- Then, we know that if Problem B is easy, Problem A must be too! But we know Problem A isn't easy, so Problem B can't be either.

Reductions I: Intuition

- **Intuition:** Problem B is "at least as hard" as problem A if we can use a black-box problem B solver (B oracle) to solve problem A with limited overhead (generally, polynomial-time).
- We know a variety of "hard" problems, so if we want to show that a problem B is hard, we need to show that oracles can be used to quickly solve some hard problem A (even if we believe that that oracle doesn't exist!).
- This is building a **reduction from** problem A **to** problem B!
- Essentially, if I want to show that Problem B is hard (and I already know that Problem A is hard), I need to assume that Problem B is easy and then solve Problem A using Problem B.
- Then, we know that if Problem B is easy, Problem A must be too! But we know Problem A isn't easy, so Problem B can't be either.

Make sure you're going in the right direction!

If you're trying to prove that a problem is NP-hard or undecidable, you need to reduce **from** an NP-hard/undecidable problem **to** the problem you want to prove is hard (in other words, show that an oracle for your problem can be used to solve an NP-hard/undecidable problem). The most common mistake on exams is reducing in the wrong direction.

Reductions II: Tutorial

- To show a problem B is NP-hard/undecidable you need to do the following:

Reductions II: Tutorial

- To show a problem B is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to (Problem B). If you're showing that something is NP-hard, you should assume that the oracle is polynomial-time.

Reductions II: Tutorial

- To show a problem B is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to (Problem B). If you're showing that something is NP-hard, you should assume that the oracle is polynomial-time.
  2. Provide an algorithm for the problem that you know is hard (Problem A) using the oracle to Problem B.

Reductions II: Tutorial

- To show a problem B is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to (Problem B). If you're showing that something is NP-hard, you should assume that the oracle is polynomial-time.
  2. Provide an algorithm for the problem that you know is hard (Problem A) using the oracle to Problem B.
  3. Analyze the runtime for your algorithm, and show it is polynomial.

Reductions II: Tutorial

- To show a problem B is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to (Problem B). If you're showing that something is NP-hard, you should assume that the oracle is polynomial-time.
  2. Provide an algorithm for the problem that you know is hard (Problem A) using the oracle to Problem B.
  3. Analyze the runtime for your algorithm, and show it is polynomial.
  4. Provide a proof of correctness.

Reductions II: Tutorial

- To show a problem B is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to (Problem B). If you're showing that something is NP-hard, you should assume that the oracle is polynomial-time.
  2. Provide an algorithm for the problem that you know is hard (Problem A) using the oracle to Problem B.
  3. Analyze the runtime for your algorithm, and show it is polynomial.
  4. Provide a proof of correctness.
- We're mostly going to be talking about **decision variants** of problems (where you only need to return YES/NO) since the main complexity classes are defined with respect to them, and since, usually, decision variants are equally hard as their calculation equivalents

Reductions II: Tutorial

- To show a problem B is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to (Problem B). If you're showing that something is NP-hard, you should assume that the oracle is polynomial-time.
  2. Provide an algorithm for the problem that you know is hard (Problem A) using the oracle to Problem B.
  3. Analyze the runtime for your algorithm, and show it is polynomial.
  4. Provide a proof of correctness.
- We're mostly going to be talking about **decision variants** of problems (where you only need to return YES/NO) since the main complexity classes are defined with respect to them, and since, usually, decision variants are equally hard as their calculation equivalents

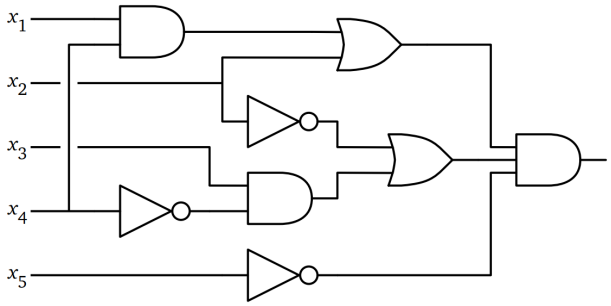
Template- Reduction

Assume that there exists an oracle function B which runs in [TIME CONSTRAINT]. Thus, we can solve A as follows:

```
1: procedure A(input):
2:     Do some preprocessing to create instances of problem B
3:     outputs ← B(generated inputs)
4:     Do some postprocessing on outputs to get the correct answer for A
```

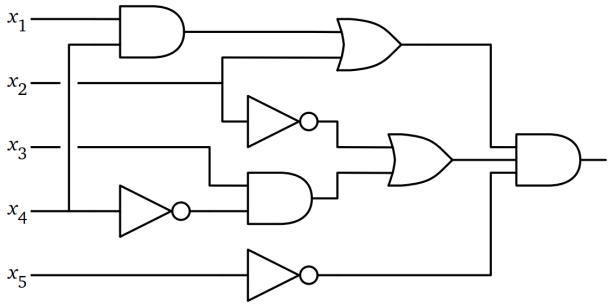
A Tour of NP-Hard Problems: CircuitSAT and 3SAT

- **CircuitSAT**: The "original" NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



A Tour of NP-Hard Problems: CircuitSAT and 3SAT

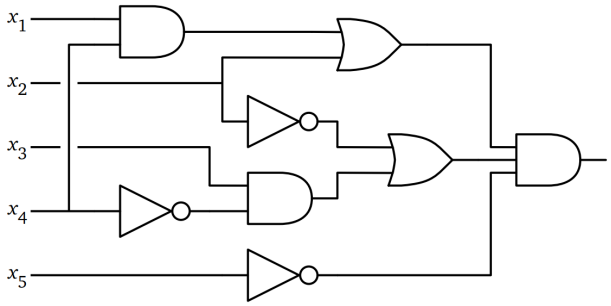
- **CircuitSAT**: The "original" NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



- **3SAT**: Given a boolean formula of the form $(a \vee b \vee c) \wedge (\overline{a} \vee d \vee e) \wedge \cdots$, is there an assignment to the input variables that makes it return `true`?

A Tour of NP-Hard Problems: CircuitSAT and 3SAT

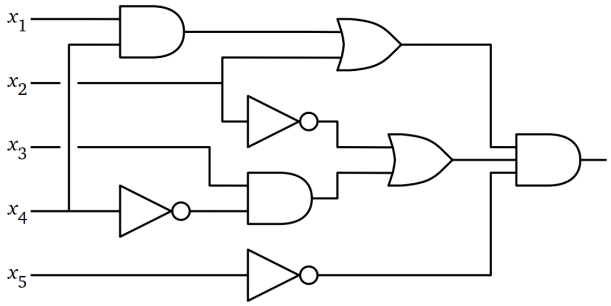
- **CircuitSAT**: The "original" NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



- **3SAT**: Given a boolean formula of the form $(a \vee b \vee c) \wedge (\overline{a} \vee d \vee e) \wedge \cdots$, is there an assignment to the input variables that makes it return `true`?
- Consider reducing from 3SAT if. . . :

A Tour of NP-Hard Problems: CircuitSAT and 3SAT

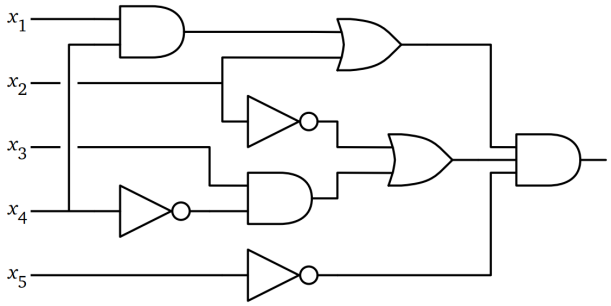
- **CircuitSAT**: The "original" NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



- **3SAT**: Given a boolean formula of the form $(a \vee b \vee c) \wedge (\overline{a} \vee d \vee e) \wedge \cdots$, is there an assignment to the input variables that makes it return `true`?
- Consider reducing from 3SAT if. . . :
  - There's some structure of choice within the problem (i.e. the goal is to decide either A or B)

A Tour of NP-Hard Problems: CircuitSAT and 3SAT

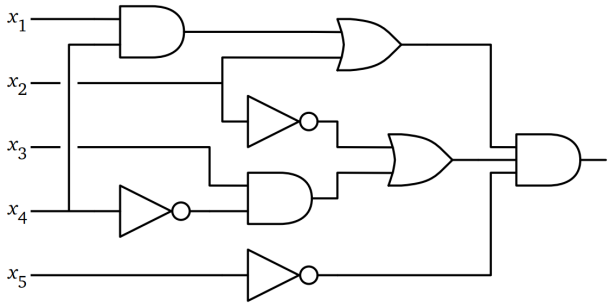
- **CircuitSAT**: The "original" NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



- **3SAT**: Given a boolean formula of the form $(a \vee b \vee c) \wedge (\overline{a} \vee d \vee e) \wedge \cdots$, is there an assignment to the input variables that makes it return `true`?
- Consider reducing from 3SAT if. . . :
  - There's some structure of choice within the problem (i.e. the goal is to decide either A or B)
  - There's a 3 in the problem, and you don't know why

A Tour of NP-Hard Problems: CircuitSAT and 3SAT

- **CircuitSAT**: The "original" NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



- **3SAT**: Given a boolean formula of the form $(a \vee b \vee c) \wedge (\overline{a} \vee d \vee e) \wedge \cdots$, is there an assignment to the input variables that makes it return `true`?
- Consider reducing from 3SAT if. . . :
  - There's some structure of choice within the problem (i.e. the goal is to decide either A or B)
  - There's a 3 in the problem, and you don't know why

Be careful with k-SAT variants!

While k-SAT for $k \geq 3$ is NP-complete, there is a polynomial-time algorithm for 2SAT. (Using strongly connected components!)

A Tour of NP-Hard Problems: CircuitSAT and 3SAT

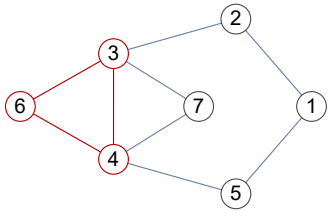
Consider the problem **MajSAT**: Clauses now consist of 5 literals, and you must satisfy at least 3 literals in each clause. Is **MajSAT** in NP, NP-hard, both, or neither? Prove why by either stating an algorithm or providing a reduction.

A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

- **MaxClique**: Given a graph G and positive integer r, can we find a K_r subgraph in G (i.e. a set of r nodes where each one has an edge to every other)?

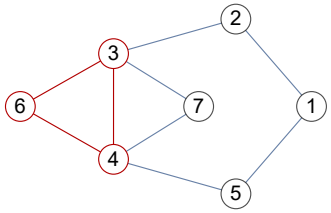
A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

- **MaxClique**: Given a graph G and positive integer r, can we find a K_r subgraph in G (i.e. a set of r nodes where each one has an edge to every other)?



A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

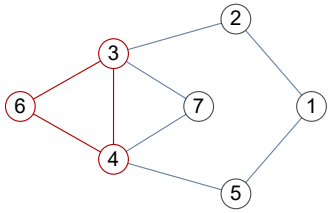
- **MaxClique**: Given a graph G and positive integer r, can we find a K_r subgraph in G (i.e. a set of r nodes where each one has an edge to every other)?



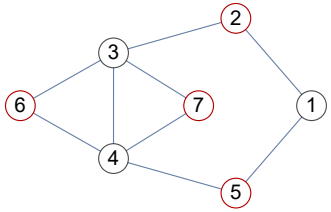
- **MaxIndSet**: Given a graph G and positive integer h, can we find a set of h nodes, none of which share an edge?

A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

- **MaxClique**: Given a graph G and positive integer r, can we find a K_r subgraph in G (i.e. a set of r nodes where each one has an edge to every other)?

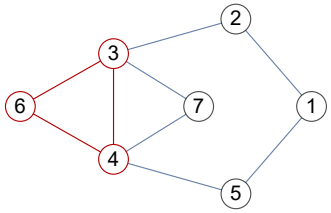


- **MaxIndSet**: Given a graph G and positive integer h, can we find a set of h nodes, none of which share an edge?

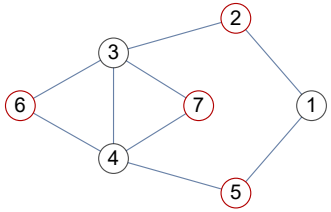


A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

- **MaxClique**: Given a graph G and positive integer r, can we find a K_r subgraph in G (i.e. a set of r nodes where each one has an edge to every other)?



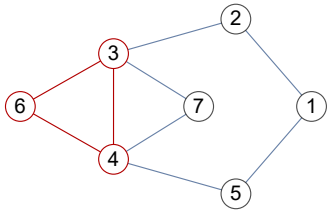
- **MaxIndSet**: Given a graph G and positive integer h, can we find a set of h nodes, none of which share an edge?



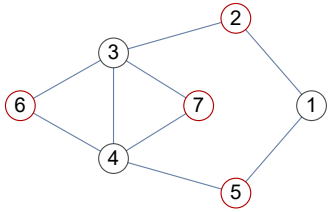
- **MinVertexCover**: Given a graph G and positive integer h, can we find a set of h nodes so that all edges have at least one endpoint chosen?

A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

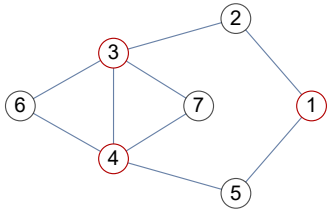
- **MaxClique**: Given a graph G and positive integer r, can we find a K_r subgraph in G (i.e. a set of r nodes where each one has an edge to every other)?



- **MaxIndSet**: Given a graph G and positive integer h, can we find a set of h nodes, none of which share an edge?



- **MinVertexCover**: Given a graph G and positive integer h, can we find a set of h nodes so that all edges have at least one endpoint chosen?

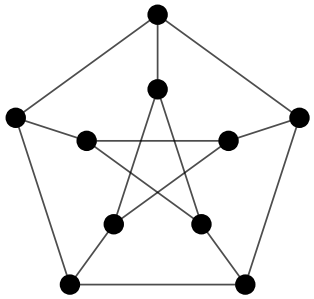


A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover, Practice

ACM is writing their review session for CS/ECE 374B MT3. While making slides, each CA writes 2 problems, either alone or in collaboration with other CAs. Since all of the CAs all have inflated egos, they won't show up to the review session unless one of the problems that they worked on is in the review session. Show that determining whether we can run a review session with at most k problems is NP-complete.

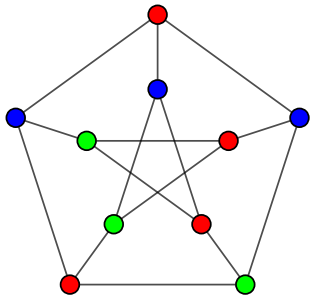
A Tour of NP-Hard Problems: Graph Coloring

- Given an (undirected) graph, can we color the nodes with at most k colors so that no two vertices that share an edge are of the same color?



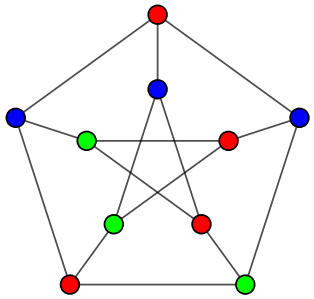
A Tour of NP-Hard Problems: Graph Coloring

- Given an (undirected) graph, can we color the nodes with at most k colors so that no two vertices that share an edge are of the same color?



A Tour of NP-Hard Problems: Graph Coloring

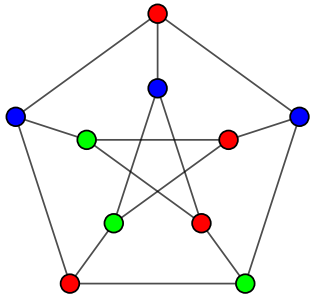
- Given an (undirected) graph, can we color the nodes with at most k colors so that no two vertices that share an edge are of the same color?



- Consider reducing from k-coloring if. . . :

A Tour of NP-Hard Problems: Graph Coloring

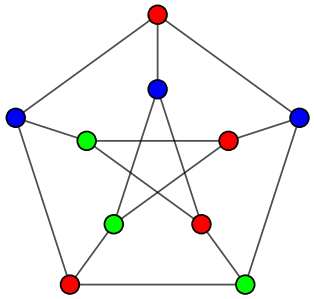
- Given an (undirected) graph, can we color the nodes with at most k colors so that no two vertices that share an edge are of the same color?



- Consider reducing from k-coloring if. . . :
  - You need to assign objects to groups, and assigning one object to a group limits your choices for some local set of others

A Tour of NP-Hard Problems: Graph Coloring

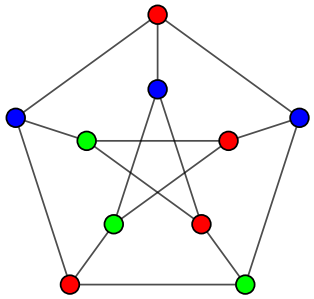
- Given an (undirected) graph, can we color the nodes with at most k colors so that no two vertices that share an edge are of the same color?



- Consider reducing from k-coloring if. . . :
  - You need to assign objects to groups, and assigning one object to a group limits your choices for some local set of others
  - There's a graph where you need to solve for some *vertex* properties

A Tour of NP-Hard Problems: Graph Coloring

- Given an (undirected) graph, can we color the nodes with at most k colors so that no two vertices that share an edge are of the same color?



- Consider reducing from k-coloring if. . . :
  - You need to assign objects to groups, and assigning one object to a group limits your choices for some local set of others
  - There's a graph where you need to solve for some *vertex* properties

Be careful with k-coloring variants!

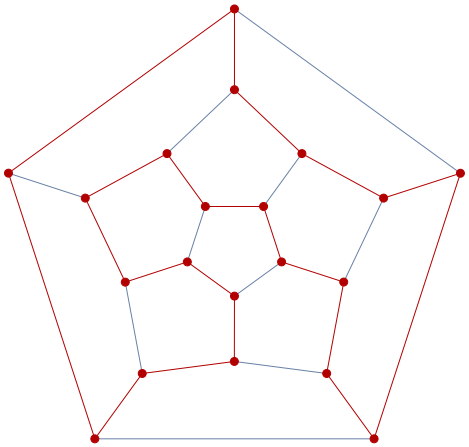
While k-coloring for $k \geq 3$ is NP-complete, you can find whether a graph is bipartite (2-colorable) using DFS.

A Tour of NP-Hard Problems: Graph Coloring: Practice

Consider the problem **Safe7Color**, which asks you to color a graph with 7 colors, such that it is a violation if there is an edge $u \leftrightarrow v$ where $c(u)$ and $c(v)$ differ by 0 or 1 (mod 7). Is this problem NP-hard?

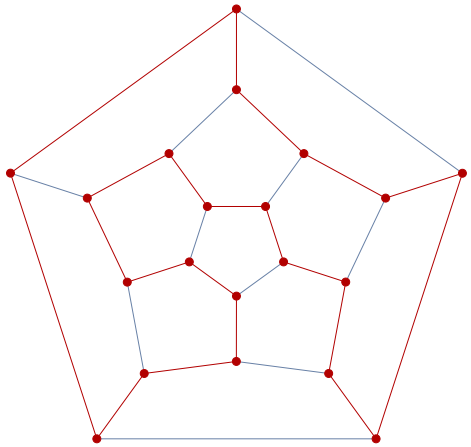
A Tour of NP-Hard Problems: Hamiltonian Paths and Cycles

- A **Hamilton Path** is a path that goes through each vertex *exactly* once. Likewise, a **Hamiltonian Cycle** is a cycle that goes through each node *exactly* once.
  - Every graph with a Hamiltonian cycle has a Hamiltonian path, but not every graph with a Hamiltonian path has a Hamiltonian cycle.



A Tour of NP-Hard Problems: Hamiltonian Paths and Cycles

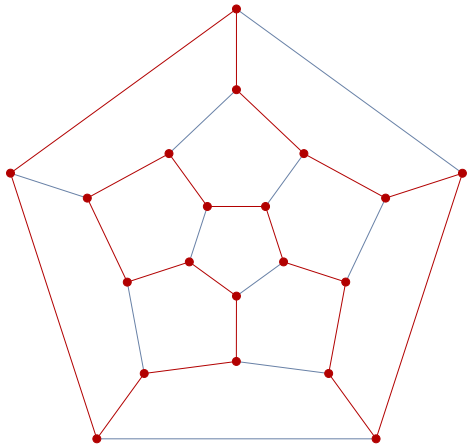
- A **Hamilton Path** is a path that goes through each vertex *exactly* once. Likewise, a **Hamiltonian Cycle** is a cycle that goes through each node *exactly* once.
  - Every graph with a Hamiltonian cycle has a Hamiltonian path, but not every graph with a Hamiltonian path has a Hamiltonian cycle.



- Consider reducing from **HamPath** or **HamCycle** if. . .

A Tour of NP-Hard Problems: Hamiltonian Paths and Cycles

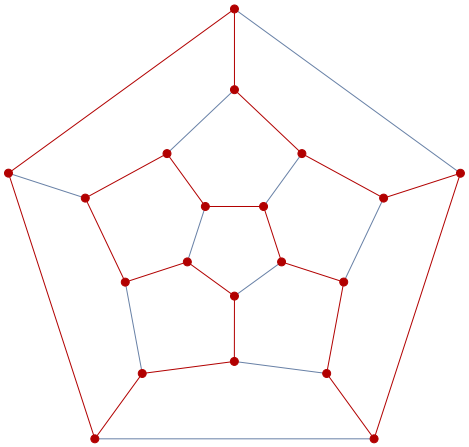
- A **Hamilton Path** is a path that goes through each vertex *exactly* once. Likewise, a **Hamiltonian Cycle** is a cycle that goes through each node *exactly* once.
  - Every graph with a Hamiltonian cycle has a Hamiltonian path, but not every graph with a Hamiltonian path has a Hamiltonian cycle.



- Consider reducing from **HamPath** or **HamCycle** if. . .
  - You're given a graph, and you're asked to find a sequence of vertices

A Tour of NP-Hard Problems: Hamiltonian Paths and Cycles

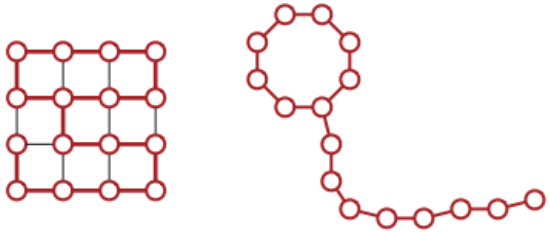
- A **Hamilton Path** is a path that goes through each vertex *exactly* once. Likewise, a **Hamiltonian Cycle** is a cycle that goes through each node *exactly* once.
  - Every graph with a Hamiltonian cycle has a Hamiltonian path, but not every graph with a Hamiltonian path has a Hamiltonian cycle.



- Consider reducing from **HamPath** or **HamCycle** if. . .
  - You're given a graph, and you're asked to find a sequence of vertices
  - You have a resource pool, and you want to use up everuthing

A Tour of NP-Hard Problems: Hamiltonian Paths and Cycles: Practice

A **balloon graph** of size ℓ is a cycle of length ℓ attached to a path of length ℓ, where the cycle and the path are disjoint except for the connecting vertex. Show that it is NP-hard to determine whether a graph has a balloon subgraph of size at least k.



NP-Hardness Practice

You are organizing a wedding between Joesphina Smith and her many husbands. Since so many families are invited, there are bound to be many interpersonal conflicts. Now you're making the seating chart, and to prevent war from breaking out, no table can have a pair of guests that despise each other. Luckily, you've spoken to Jeff (the gossip of the town), who told you which pairs of guests hate each other. Prove that finding the minimum number of tables needed to peacefully seat everyone is NP-Hard.

NP-Hardness Practice

h

NP-Hardness Practice

h

NP-Hardness Practice: T/F

h

Undecidability

- A language is **decidable** if there exists an algorithm which always returns `true` to all inputs in L and `false` to inputs not in L
  - If we can only return `true` to all inputs in L and either return `false` *or* infinite-loop for all other inputs, the language is merely **acceptable**.

Theorem (Turing, 1936)

The language `Halt`$: \{(f, w) :$ *the function* f *does not infinite loop on input* $w\}$ *is undecidable.*

- 3 main ways to prove that a problem is undecidable:

Undecidability

- A language is **decidable** if there exists an algorithm which always returns `true` to all inputs in L and `false` to inputs not in L
  - If we can only return `true` to all inputs in L and either return `false` *or* infinite-loop for all other inputs, the language is merely **acceptable**.

Theorem (Turing, 1936)

The language `Halt`*:* $\{(f, w) :$ *the function* f *does not infinite loop on input* $w\}$ *is undecidable.*

- 3 main ways to prove that a problem is undecidable:
  1. Reduce from Halt: Given an oracle for your problem, design an algorithm to decide Halt. No runtime requirement!

Undecidability

- A language is **decidable** if there exists an algorithm which always returns `true` to all inputs in L and `false` to inputs not in L
  - If we can only return `true` to all inputs in L and either return `false` *or* infinite-loop for all other inputs, the language is merely **acceptable**.

Theorem (Turing, 1936)

The language `Halt`$: \{(f, w) :$ *the function* f *does not infinite loop on input* $w\}$ *is undecidable.*

- 3 main ways to prove that a problem is undecidable:
  1. Reduce from Halt: Given an oracle for your problem, design an algorithm to decide Halt. No runtime requirement!
  2. Rice's Theorem: Very powerful, basically claims that any non-trivial question about functions/Turing machines is undecidable.

Undecidability

- A language is **decidable** if there exists an algorithm which always returns `true` to all inputs in L and `false` to inputs not in L
  - If we can only return `true` to all inputs in L and either return `false` *or* infinite-loop for all other inputs, the language is merely **acceptable**.

Theorem (Turing, 1936)

The language `Halt`$: \{(f, w) :$ *the function* f *does not infinite loop on input* $w\}$ *is undecidable.*

- 3 main ways to prove that a problem is undecidable:
  1. Reduce from Halt: Given an oracle for your problem, design an algorithm to decide Halt. No runtime requirement!
  2. Rice's Theorem: Very powerful, basically claims that any non-trivial question about functions/Turing machines is undecidable.
  3. Abuse the fact that you can put code into a function to derive a contradiction.

Theorem (Rice)

Let $\mathcal{L}$ *be any set of languages that satisfies the following conditions:*

- *There is a Turing machine* Y *such that Accept(*Y*)* $\in \mathcal{L}$ *.*
- *There is a Turing machine* N *such that Accept(*N*)* $\notin \mathcal{L}$ *.*

Then, the language AcceptIn$(\mathcal{L}) \leftarrow \{\langle M \rangle \mid$ *Accept*$(M) \in \mathcal{L}\}$ *is undecidable.*

it's hard to decide

For each of the following languages, either show that they are decidable by describing an algorithm that decides them, or show that they are undecidable by reduction and by Rice's theorem when possible.

(a) AcceptsRegular = $\{\langle M \rangle : M$'s accept set is regular$\}$

it's hard to decide

For each of the following languages, either show that they are decidable by describing an algorithm that decides them, or show that they are undecidable by reduction and by Rice's theorem when possible.

(a) AcceptsRegular $= \{\langle M \rangle : M$'s accept set is regular$\}$

(b) HaltsQuadratically
$= \{(\langle M \rangle, r) : M$ halts on r in at most $|r|^2$ arithmetic operations$\}$

it's hard to decide

For each of the following languages, either show that they are decidable by describing an algorithm that decides them, or show that they are undecidable by reduction and by Rice's theorem when possible.

(a) AcceptsRegular $= \{\langle M \rangle : M$'s accept set is regular$\}$

(b) HaltsQuadratically
$= \{(\langle M \rangle, r) : M$ halts on r in at most $|r|^2$ arithmetic operations$\}$

(c) AcceptsRejects $= \{\langle M \rangle : M$'s accept set $= M$'s reject set$\}$

it's hard to decide

For each of the following languages, either show that they are decidable by describing an algorithm that decides them, or show that they are undecidable by reduction and by Rice's theorem when possible.

(a) AcceptsRegular = $\{\langle M \rangle : M$'s accept set is regular$\}$

(b) HaltsQuadratically
$= \{(\langle M \rangle, r) : M$ halts on r in at most $|r|^2$ arithmetic operations$\}$

(c) AcceptsRejects = $\{\langle M \rangle : M$'s accept set = M's reject set$\}$

(d) CFLAccepts374 = $\{c \in CFGs : c$ accepts exactly 374 strings$\}$

it's hard to decide

For each of the following languages, either show that they are decidable by describing an algorithm that decides them, or show that they are undecidable by reduction and by Rice's theorem when possible.

(a) AcceptsRegular $= \{\langle M \rangle : M$'s accept set is regular$\}$

(b) HaltsQuadratically
$= \{(\langle M \rangle, r) : M$ halts on r in at most $|r|^2$ arithmetic operations$\}$

(c) AcceptsRejects $= \{\langle M \rangle : M$'s accept set $= M$'s reject set$\}$

(d) CFLAccepts374 $= \{c \in CFGs : c$ accepts exactly 374 strings$\}$

(e) LeftThrice $= \{\langle M, w \rangle : M$ moves left on input w three times in a row$\}$

it's hard to decide

For each of the following languages, either show that they are decidable by describing an algorithm that decides them, or show that they are undecidable by reduction and by Rice's theorem when possible.

(a) AcceptsRegular $= \{\langle M \rangle : M$'s accept set is regular$\}$

(b) HaltsQuadratically
$= \{(\langle M \rangle, r) : M$ halts on r in at most $|r|^2$ arithmetic operations$\}$

(c) AcceptsRejects $= \{\langle M \rangle : M$'s accept set $= M$'s reject set$\}$

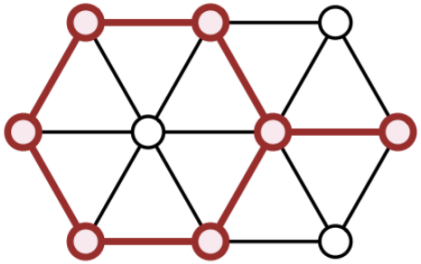
(d) CFLAccepts374 $= \{c \in CFGs : c$ accepts exactly 374 strings$\}$

(e) LeftThrice $= \{\langle M, w \rangle : M$ moves left on input w three times in a row$\}$

(f) NeverLeft $= \{\langle M, w \rangle : M$ never moves left on input $w\}$

twiangles are scawy

A subset S of vertices in an undirected graph G is a called triangle-free if, for every triple of vertices $u, v, w \in S$,at least one of the three edges uv, uw, vw is absent from G. Prove that finding the size of the largest triangle-free subset of vertices in a given undirected graph is NP-hard.



A triangle-free subset of 7 vertices and its induced edges.
This is ***not*** the largest triangle-free subset in this graph.

Feedback

- Please fill out the feedback form:
  `go.acm.illinois.edu/374a_final_feedback`