CS 374A Final Review

The Multiple-Choice Final Boss of Algorithms

ACM @ UIUC

November 3, 2024

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 Kani'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 Kani'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* <OBJECT>. Assume for all k s.t. k is smaller than x (by <ORDERING PROPERTY>), that $P(k)$ holds.

If $x =$ <MINIMAL OBJECT>, then . . . , so $P(x)$ holds

If $x \neq$ <MINIMAL OBJECT>, then . . . , so by IH, . . . , so $P(x)$ holds.

Thus, in all cases, $P(x)$ holds.

Induction

Template

Let x be an *arbitrary* <OBJECT>. Assume for all k s.t. k is smaller than x (by <ORDERING PROPERTY>), that $P(k)$ holds.

If $x =$ <MINIMAL OBJECT>, then ..., so $P(x)$ holds

If $x \neq$ <MINIMAL OBJECT>, then ..., so by IH, ..., so $P(x)$ holds.

Thus, in all cases, $P(x)$ holds.

Some tips:

- **Always use strong induction**. All weak inductive proofs can be re-written to use strong induction with minimal changes, and the extra assumption can make your life significantly easier.
- **Write out your IH, base case, and inductive step out explicitly.** Doing so will help you avoid getting confused, and will help you avoid losing points.
- If you're performing induction on a recursive definition (strings, CFLs, etc.), generally, your inductive step will consist of one step of the recursion, and then will use IH.

Regular Languages/Expressions

- Built inductively on 3 operations:

Regular Languages/Expressions

- Built inductively on 3 operations:
  - $+$ is the union operator. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$

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. $L(r_1^*) = L(r_1)^*$

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

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})

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. When processing a string through a DFA, you only need to know which state you're currently in, and do not need to look forwards/backwards in the string.
  - Implementing a DFA/NFA in code only requires $O(1)$ memory
  - If your checker program needs to count something without bound, the language you're checking isn't 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. When processing a string through a DFA, you only need to know which state you're currently in, and do not need to look forwards/backwards in the string.
  - Implementing a DFA/NFA in code only requires $O(1)$ memory
  - If your checker program needs to count something without bound, the language you're checking isn't regular.
- **Regex Design Tips:** If you don't know where to start, try giving examples for strings that are in the language and strings that aren't. Look for patterns and try to build components around those patterns, then combine into something that represents the full language. Make sure to test and modify for edge cases. Explain, in English, each part of your regular expression with a short sentence. Does the explanation match the language?

DFAs/NFAs

- DFAs defined by *state set* Q, *accepting set* $A \subseteq Q$, *input alphabet* Σ, *start state* $s \in Q$, and *transition function* $\delta : Q \times \Sigma \to Q$
- NFAs allow for “trying” multiple transitions at the same time or transitioning without reading in (ϵ-transitions), accepts if there is a path to an accepting state. Transition function thereby changes to $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$
  - Power-set construction to convert from NFA to DFA- in theory exponential-time but used in practice.
- **Tips for creating DFA/NFAs**: Break down your language into smaller patterns, and figure out what you need to store as state for each part. Make sure you clearly define all components. A drawing or transition table is just as valid as a $(Q, A, \Sigma, s, \delta)$ definition.

DFAs/NFAs

- DFAs defined by *state set* Q, *accepting set* $A \subseteq Q$, *input alphabet* Σ, *start state* $s \in Q$, and *transition function* $\delta : Q \times \Sigma \to Q$
- NFAs allow for "trying" multiple transitions at the same time or transitioning without reading in (ϵ-transitions), accepts if there is a path to an accepting state. Transition function thereby changes to $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$
  - Power-set construction to convert from NFA to DFA- in theory exponential-time but used in practice.
- **Tips for creating DFA/NFAs**: Break down your language into smaller patterns, and figure out what you need to store as state for each part. Make sure you clearly define all components. A drawing or transition table is just as valid as a $(Q, A, \Sigma, s, \delta)$ definition.

Product Constructions

Given some languages $L_1, \ldots, L_n$ we want a DFA that accepts strings w satisfying $f(w \in L_1, \ldots, w \in L_n)$ where f is some logical function. Create a DFA/NFA for L using the following *rough* format:

- $Q = Q_1 \times \cdots \times Q_n$
- $\delta'(q_1, \ldots, q_n) = (\delta_1(q_1), \ldots, \delta_2(q_2))$
- $s = (s_1, \ldots, s_n)$
- $A' = \{$convert f into a set expression$\}$

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 w, w', and we know that there exists a **distinguishing suffix** z s.t. $wz \in L, w'z \notin L$, then w, w' 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 w, w', and we know that there exists a **distinguishing suffix** z s.t. $wz \in L, w'z \notin L$, then w, w' 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

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 w, w', and we know that there exists a **distinguishing suffix** z s.t. $wz \in L, w'z \notin L$, then w, w' 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 w, w', and we know that there exists a **distinguishing suffix** z s.t. $wz \in L, w'z \notin L$, then w, w' 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
- If you see the need to keep track of something without bound, you can create a fooling set around the part where you count up.

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 w, w', and we know that there exists a **distinguishing suffix** z s.t. $wz \in L, w'z \notin L$, then w, w' 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
- If you see the need to keep track of something without bound, you can create a fooling set around the part where you count up.
- **If you see divisibility, think primes!** All primes are coprime, so primality provides for an infinite set with easier construction of distinguishing suffixes.

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 w, w', and we know that there exists a **distinguishing suffix** z s.t. $wz \in L, w'z \notin L$, then w, w' 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
- If you see the need to keep track of something without bound, you can create a fooling set around the part where you count up.
- **If you see divisibility, think primes!** All primes are coprime, so primality provides for an infinite set with easier construction of distinguishing suffixes.
- If you're using strings of the form $1^k, 0^p$, etc. when sampling elements of your fooling set a^i, a^j, it's completely fine to assume WLOG that $j > i$, but nothing about the underlying structure of i and j. If you want to put in such a restriction, you should instead restrict your fooling set further.

Language Transforms

- Used to prove that regularity is closed under some function f (if L is regular, then $f(L)$ is regular).
- **General Format:** Given a DFA M that accepts L, create an NFA M' that accepts $f(L)$.

Language Transforms

- Used to prove that regularity is closed under some function f (if L is regular, then $f(L)$ is regular).
- **General Format:** Given a DFA M that accepts L, create an NFA M' that accepts $f(L)$.
- **General Strategy:** Apply non-determinism to guess the future behavior of the DFA that you want to simulate.

Language Transforms

- Used to prove that regularity is closed under some function f (if L is regular, then $f(L)$ is regular).
- **General Format:** Given a DFA M that accepts L, create an NFA M' that accepts $f(L)$.
- **General Strategy:** Apply non-determinism to guess the future behavior of the DFA that you want to simulate.

Make sure you're going in the right direction!

If you see the format $f(L) = \{k(w) : w \in L\}$, your modified NFA should be trying to *undo* k, while if you see the format $f(L) = \{w : k(w) \in L\}$, your modified NFA should be trying to *apply* k. Mixing these up is the most common mistake we see on homeworks/exams.

In some cases, only one direction is possible. For example, $\texttt{un-palin}(L) : \{w : ww^R \in L\}$ has a transformation construction, but $\texttt{palin}(L) = \{ww^R : w \in L\}$ is irregular for some L.

Context-Free Languages/Grammars

- Formally, a context-free grammar is defined by *nonterminals/variables* V, *terminals/symbols* T, *productions* P, and the *start symbol* S. Each production rule in P looks like $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$.

Context-Free Languages/Grammars

- Formally, a context-free grammar is defined by *nonterminals/variables* V, *terminals/symbols* T, *productions* P, and the *start symbol* S. Each production rule in P looks like $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$.
- For example, consider $V = \{S\}$, $T = \{0, 1\}$, $P = \{S \to \epsilon, S \to 0S1\}$. (You can abbreviate this to $P = \{S \to \epsilon \mid 0S1\}$.) What language is this?

Intuition

CFGs "build" strings, going from the outside in; you can choose rules to add characters on the left/right.
Alternatively, CFGs "peel back" strings, removing characters from the left/right until nothing is left.

Context-Free Languages/Grammars

- Formally, a context-free grammar is defined by *nonterminals/variables* V, *terminals/symbols* T, *productions* P, and the *start symbol* S. Each production rule in P looks like $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$.
- For example, consider $V = \{S\}$, $T = \{0, 1\}$, $P = \{S \to \epsilon, S \to 0S1\}$. (You can abbreviate this to $P = \{S \to \epsilon \mid 0S1\}$.) What language is this?

Intuition

CFGs "build" strings, going from the outside in; you can choose rules to add characters on the left/right.
Alternatively, CFGs "peel back" strings, removing characters from the left/right until nothing is left.

- CFLs only closed under union, kleene star, and concatenation. CFLs are *not* closed under intersection or complement.

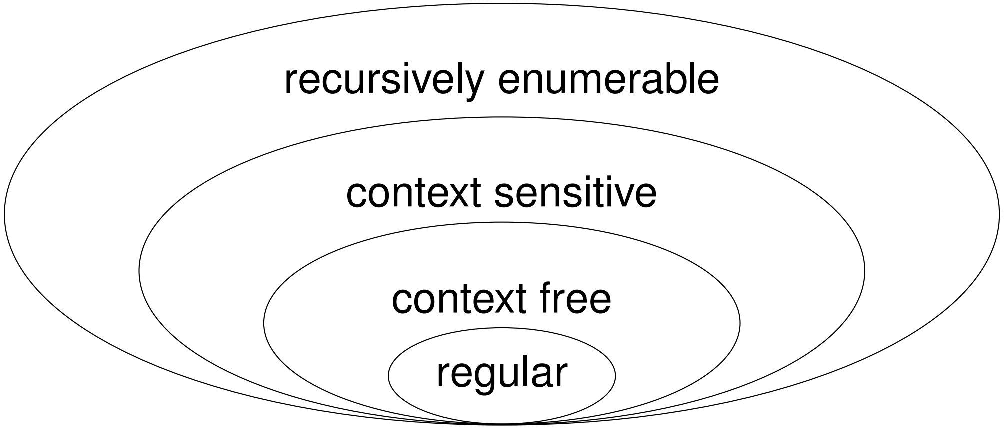
Context-Free Languages/Grammars

- Formally, a context-free grammar is defined by *nonterminals/variables* V, *terminals/symbols* T, *productions* P, and the *start symbol* S. Each production rule in P looks like $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$.
- For example, consider $V = \{S\}$, $T = \{0, 1\}$, $P = \{S \to \epsilon, S \to 0S1\}$. (You can abbreviate this to $P = \{S \to \epsilon \mid 0S1\}$.) What language is this?

Intuition

CFGs "build" strings, going from the outside in; you can choose rules to add characters on the left/right.
Alternatively, CFGs "peel back" strings, removing characters from the left/right until nothing is left.

- CFLs 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, $L \cap \{0^n1^n : n > 0\}$ 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 $c_i = \begin{cases} 1 & \text{if } \exists n \in \mathbf{Z} \text{ s.t. } i = 2^n \\ c_i & \text{otherwise} \end{cases}$.
   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 $c_i = \begin{cases} 1 & \text{if } \exists n \in \mathbf{Z} \text{ s.t. } i = 2^n \\ c_i & \text{otherwise} \end{cases}$.
   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

4. If L is decided by a DFA with n states, then consider the language L' consisting of all strings in L with at most 374 characters removed. Which of the following is true?
   (a) L' can be decided by a DFA with $O(n)$ states
   (b) L' can be decided by a DFA whose number of states is polynomial in n
   (c) L' can be decided by a DFA whose number of states is exponential in n
   (d) We cannot guarantee that there exists a DFA which decides L'

Practice Questions II

4. If L is decided by a DFA with n states, then consider the language L' consisting of all strings in L with at most 374 characters removed. Which of the following is true?
   (a) L' can be decided by a DFA with $O(n)$ states
   (b) L' can be decided by a DFA whose number of states is polynomial in n
   (c) L' can be decided by a DFA whose number of states is exponential in n
   (d) We cannot guarantee that there exists a DFA which decides L'
5. 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. If L is decided by a DFA with n states, then consider the language L' consisting of all strings in L with at most 374 characters removed. Which of the following is true?
   (a) L' can be decided by a DFA with $O(n)$ states
   (b) L' can be decided by a DFA whose number of states is polynomial in n
   (c) L' can be decided by a DFA whose number of states is exponential in n
   (d) We cannot guarantee that there exists a DFA which decides L'
5. 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.
6. 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

7. 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

7. 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
8. 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 III

7. 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
8. 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
9. Given a regular expression of length n, the equivalent minimum DFA might have
   9.1 $O(n)$ states
   9.2 $O(n^2)$ states
   9.3 $O(2^n)$ states
   9.4 $O(n!)$ states
   9.5 Always exactly n states

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: Tower of Hanoi

Template

```
procedure AmazingRecursiveAlgo(n)
    if n == [some base case] then
        return [value]
    else
        return AmazingRecursiveAlgo(n − 1)
```

- 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 behaviour 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 behaviour of the algorithm at hand.
- **Master's Theorem**

Master's Theorem

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

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 behaviour of the algorithm at hand.
- **Master's Theorem**

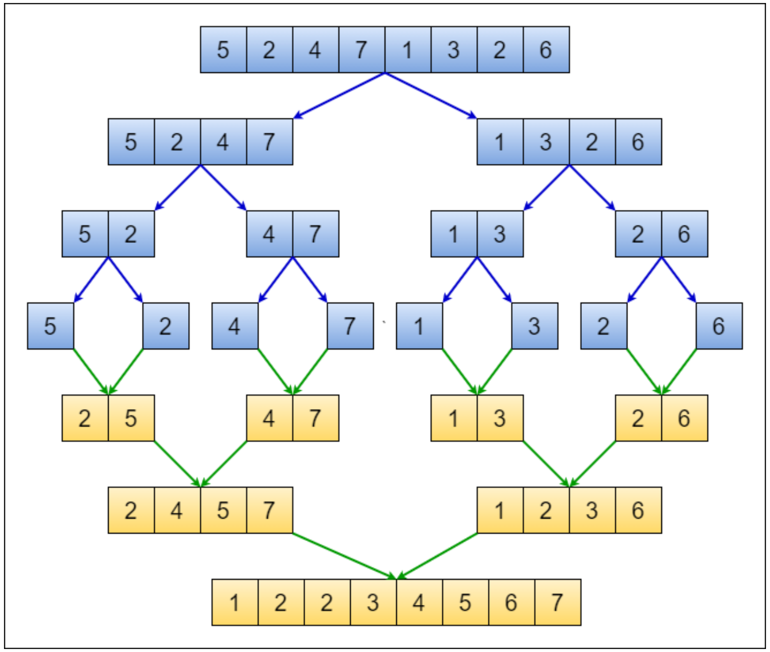
Master's Theorem

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

  - **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.

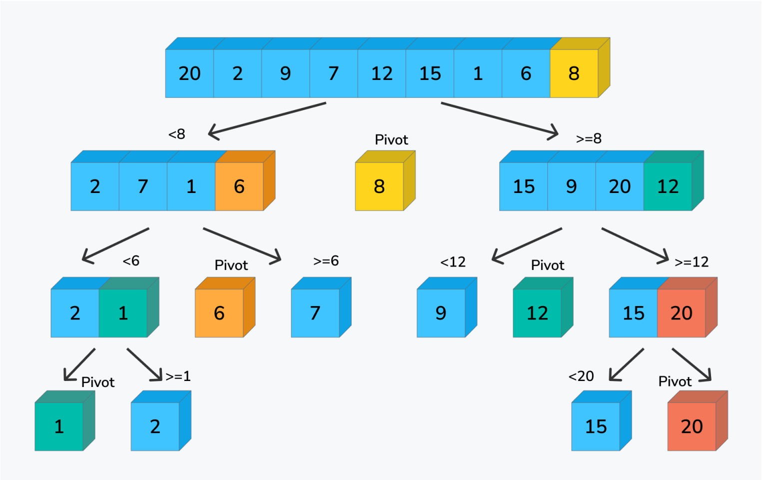


Divide and Conquer Algos: Quickselect

- **Purpose:** Get the n^{th} smallest element in an arbitrary array.
- **Time Complexity**: Avg: $O(n)$ | Worst; $O(n^2)$, ($O(n)$ with MoM)
- **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, just like quick select. If the length of the array of elements that are less than P_V is greater than n, then we know that the n^{th} smallest element is to the left of P and we recurse on the left subarray. Otherwise, we know that the n^{th} smallest element is to the right of P and we recurse on the right subarray.
  - **Why the poor worst case performance?**
  - Again, because we can get unlucky and pick the worst possible pivot at every step.
  - We can guarantee linear performance with a better pivot-picking algorithm such as MEDIANOFMEDIANS
    - Finds element that larger than $\frac{3}{10}$ and smaller than $\frac{7}{10}$ of the array's elements.
    - Runs in $O(n)$ time

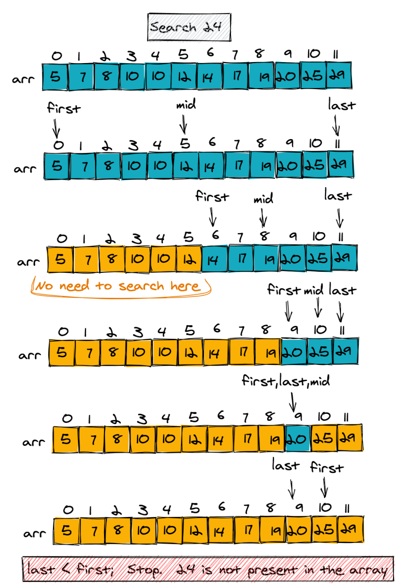
Divide and Conquer Algos: Quicksort

- **Purpose:** Sort an arbitrary array.
- **Time Complexity**: Avg: $O(n \log n)$ | Worst: $O(n^2)$ ($O(n \log n)$ deterministic with quickselect partitioning)
- **Intuition**: Pick a pivot and rearrange the array such that all the elements that are less than the pivot value are to the left of the pivot value and all the elements that are greater than the pivot value are to the right of the pivot value. Then sort each side.
  - **Why the poor worst case performance?**
  - Because we can get unlucky and pick the worst possible pivot at every step.



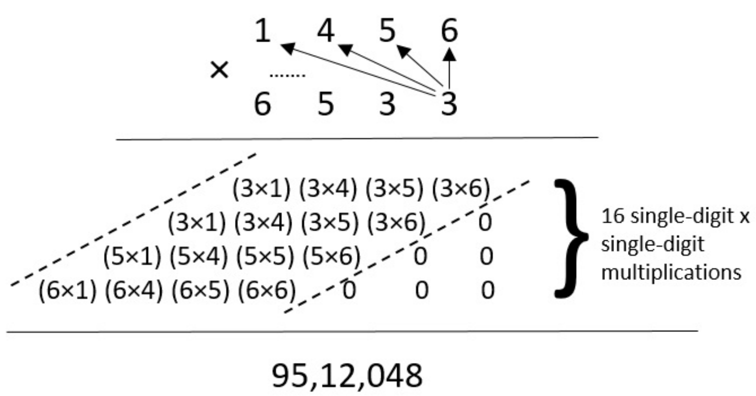
Divide and Conquer Algos: Binary Search

- **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 n. Pick the middle element M in the array. If $n > M$, the element must be to the right of n 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.



Backtracking

- Technique to methodically explore the solutions to a problem via the reduction to said problem to a smaller variant of itself, a.k.a **recursion**.
- Intuitively, think of the problem space as a maze that we are trying to find the exit of. For each path, you would traverse until you reach a dead end, at which point you **back track** to try a different path.
- To find recurrence, think ”What information about a subset of my current problem space would be really nice to know?”

Backtracking

- Technique to methodically explore the solutions to a problem via the reduction to said problem to a smaller variant of itself, a.k.a **recursion**.
- Intuitively, think of the problem space as a maze that we are trying to find the exit of. For each path, you would traverse until you reach a dead end, at which point you **back track** to try a different path.
- To find recurrence, think ”What information about a subset of my current problem space would be really nice to know?” **Example:** Longest Increasing Subsequence
- ”What is the length of a longest increasing subsequence in an arbitrary array?”

Backtracking

- Technique to methodically explore the solutions to a problem via the reduction to said problem to a smaller variant of itself, a.k.a **recursion**.
- Intuitively, think of the problem space as a maze that we are trying to find the exit of. For each path, you would traverse until you reach a dead end, at which point you **back track** to try a different path.
- To find recurrence, think ”What information about a subset of my current problem space would be really nice to know?” **Example:** Longest Increasing Subsequence
- ”What is the length of a longest increasing subsequence in an arbitrary array?”

$$\mathrm{LIS}(i,j) = \begin{cases} 0 & \text{if } i = 0 \\ \mathrm{LIS}(i-1,j) & \text{if } A[i] \geq A[j] \\ \max \begin{cases} \mathrm{LIS}(i-1,j) \\ 1 + \mathrm{LIS}(i-1,i) \end{cases} & \text{else} \end{cases}$$

Backtracking

- Technique to methodically explore the solutions to a problem via the reduction to said problem to a smaller variant of itself, a.k.a **recursion**.
- Intuitively, think of the problem space as a maze that we are trying to find the exit of. For each path, you would traverse until you reach a dead end, at which point you **back track** to try a different path.
- To find recurrence, think "What information about a subset of my current problem space would be really nice to know?" **Example:** Longest Increasing Subsequence
- "What is the length of a longest increasing subsequence in an arbitrary array?"

$$\mathrm{LIS}(i,j) = \begin{cases} 0 & \text{if } i = 0 \\ \mathrm{LIS}(i-1,j) & \text{if } A[i] \geq A[j] \\ \max \begin{cases} \mathrm{LIS}(i-1,j) \\ 1 + \mathrm{LIS}(i-1,i) \end{cases} & \text{else} \end{cases}$$

This kind of sucks; we're redoing computation that we've already done! What if instead, we computed all the subproblems beforehand, wrote down the solutions, then did the recursion?

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

Let’s look at the LIS example from before: ”What is the length of a longest increasing subsequence in an arbitrary array?”

Dynamic Programming

Let's look at the LIS example from before: ”What is the length of a longest increasing subsequence in an arbitrary array?”

procedure LIS-ITERATIVE(A[1..n]):
- $A \leftarrow [1 \dots n][1 \dots n]$
- **for all** $i \leftarrow 1 \dots n$ **do**
  - **for all** $j \leftarrow i \dots n$ **do**
    - **if** $A[i] \leq A[j]$ **then**
      - $LIS[i][j] = 1$
    - **else**
      - $LIS[i][j] = 0$
- **for all** $i \leftarrow 1 \dots n$ **do**
  - **for all** $j \leftarrow 2 \dots n$ **do**
    - **if** $A[i] \geq A[j]$ **then**
      - $LIS[i][j] = LIS[i-1, j]$
    - **else**
      - $$LIS[i][j] = \max \begin{cases} LIS[i-1, j] \\ LIS[i-1, i] + 1 \end{cases}$$
- **return** LIS[n, n]

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 $u \to v$

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 $u \to v$
  - **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 $u \to v$
  - **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.

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 $u \to v$
  - **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.
- **Connected:** $u, v \in V$ are connected $\iff$ there exists a path between u and 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 $u \to v$
  - **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.
- **Connected:** $u, v \in V$ are connected $\iff$ there exists a path between u and v.
- **Strongly Connected:** $u, v \in V$ are strongly connected $\iff$ there exists a path between u and v and from v to u.

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 $u \to v$
  - **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.
- **Connected:** $u, v \in V$ are connected $\iff$ there exists a path between u and v.
- **Strongly Connected:** $u, v \in V$ are strongly connected $\iff$ there exists a path between u and v and from v to u.
- **Connected Component (of u):** The set of all vertices connected to u.

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.
- **Connected:** $u, v \in V$ are connected $\iff$ there exists a path between u and v.
- **Strongly Connected:** $u, v \in V$ are strongly connected $\iff$ there exists a path between u and v and from v to u.
- **Connected Component (of u):** The set of all vertices connected to u.
- **Strongly Connected Component:** A set of vertices a strongly connected component if each pair of vertices are strongly connected.

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 $u \to v$
  - **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.
- **Connected:** $u, v \in V$ are connected $\iff$ there exists a path between u and v.
- **Strongly Connected:** $u, v \in V$ are strongly connected $\iff$ there exists a path between u and v and from v to u.
- **Connected Component (of u):** The set of all vertices connected to u.
- **Strongly Connected Component:** A set of vertices a strongly connected component if each pair of vertices are strongly connected.

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 **Quake Heaps**, $O(m + n \log n)$)

Graph Algorithms: Shortest Path

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

- **Dijkstra's**
  - **Purpose:** SSSP, no negative edges
  - **Implementation:** Visit neighbours in **priority queue**
  - **Runtime:** $O(m \log n)$ (with **Quake Heaps**, $O(m + n \log n)$)
- **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:** $O(mn)$
- **Floyd-Warshall**
  - **Purpose:** APSP, yes negative edge weights
  - **Implementation:** Dynamic Programming recurrence:
    - $d(u, v, i)$ is the shortest-path distance from u to v only going through vertices $1 \ldots i$.
    - $d(u, v, i) = \min \left(d(u, v, i), d(u, i, i-1) + d(i, v, i-1) \right)$
  - **Runtime:** $O(n^3)$

Graph Algorithms: MSTs

3 main algorithms:

- **Prim-Jarnik**: Keep a priority queue for edges to be added to the tree. Start the tree at some arbitrarily selected root vertex. When adding a vertex, add all of its neighbors to the queue. Runtime: $O(|E| \log |V|)$, $O(|V| \log |V| + |E|)$ using Quake heaps.

Graph Algorithms: MSTs

3 main algorithms:

- **Prim-Jarnik**: Keep a priority queue for edges to be added to the tree. Start the tree at some arbitrarily selected root vertex. When adding a vertex, add all of its neighbors to the queue. Runtime: $O(|E| \log |V|)$, $O(|V| \log |V| + |E|)$ using Quake heaps.
- **Kruskal**: Keep a disjoint-sets data structure to keep track of connected components. Sort the edges, then in order, add each edge if it connects two components. Runtime: $O(|E| \log |V|)$.

Graph Algorithms: MSTs

3 main algorithms:

- **Prim-Jarnik**: Keep a priority queue for edges to be added to the tree. Start the tree at some arbitrarily selected root vertex. When adding a vertex, add all of its neighbors to the queue. Runtime: $O(|E| \log |V|)$, $O(|V| \log |V| + |E|)$ using Quake heaps.
- **Kruskal**: Keep a disjoint-sets data structure to keep track of connected components. Sort the edges, then in order, add each edge if it connects two components. Runtime: $O(|E| \log |V|)$.
- **Borůvka**: No fancy data structures! Just find smallest edge going out of each vertex, then contract all edges that you selected! Runtime: $O(|E| \log |V|)$

Graph Algorithms: MSTs

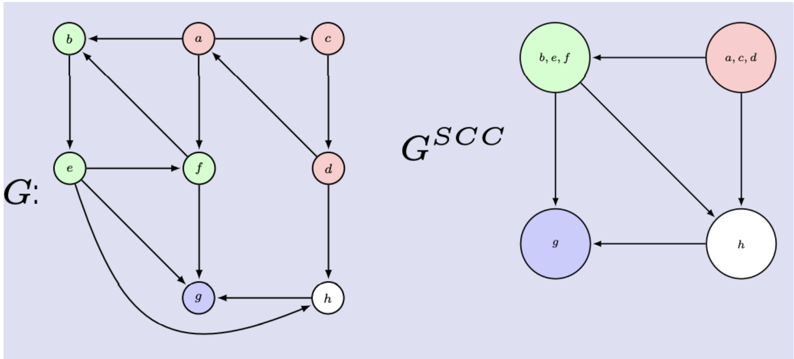
3 main algorithms:

- **Prim-Jarnik**: Keep a priority queue for edges to be added to the tree. Start the tree at some arbitrarily selected root vertex. When adding a vertex, add all of its neighbors to the queue. Runtime: $O(|E| \log |V|)$, $O(|V| \log |V| + |E|)$ using Quake heaps.
- **Kruskal**: Keep a disjoint-sets data structure to keep track of connected components. Sort the edges, then in order, add each edge if it connects two components. Runtime: $O(|E| \log |V|)$.
- **Borůvka**: No fancy data structures! Just find smallest edge going out of each vertex, then contract all edges that you selected! Runtime: $O(|E| \log |V|)$
- Faster (but way more complicated algorithms) exist. **Yao** (1975): $O(|E| \log \log |V|)$ with a modification of Borůvka's (using linear-time median selection). **Karger-Klein-Tarjan** (1995): $O(|E|)$ in expectation, **Chazelle** (2000): $O(|E|\alpha(|V|, |E|))$ deterministic

Graph Algorithms: SCC

SCC-Finding Algorithms (Tarjan's, Kosaraju's)

- **Purpose:** To identify (and collapse) SCCs in a (directed) graph
- **Runtime:** $O(V + E)$
- **Returns:** A metagraph that has one node for each SCC.



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.

Practice Problems I

1. Given a graph, consider the problem of finding the vertex that is reachable by the most other vertices. This problem is:
   (a) Solvable in $O(m+n)$ time
   (b) Solvable in $O(m \log n)$ time
   (c) Solvable in $O(mn)$ time
   (d) Solvable in polynomial time, but not any of the runtimes above
   (e) Solvable in exponential time, but not polynomial time

Practice Problems I

1. Given a graph, consider the problem of finding the vertex that is reachable by the most other vertices. This problem is:
   (a) Solvable in $O(m + n)$ time
   (b) Solvable in $O(m \log n)$ time
   (c) Solvable in $O(mn)$ time
   (d) Solvable in polynomial time, but not any of the runtimes above
   (e) Solvable in exponential time, but not polynomial time
2. Given an unsorted list, we want to print out the $\sqrt{n}$ smallest elements in sorted order.
   (a) We can do this in $O(\sqrt{n})$ operations
   (b) We can do this in $O(\sqrt{n} \log n)$ operations
   (c) We can do this in $O(n)$ operations
   (d) We can do this in $O(n \log n)$ operations
   (e) We can do this in $O(n^{1.5})$ operations

Practice Problems I

1. Given a graph, consider the problem of finding the vertex that is reachable by the most other vertices. This problem is:
   (a) Solvable in $O(m+n)$ time
   (b) Solvable in $O(m \log n)$ time
   (c) Solvable in $O(mn)$ time
   (d) Solvable in polynomial time, but not any of the runtimes above
   (e) Solvable in exponential time, but not polynomial time
2. Given an unsorted list, we want to print out the $\sqrt{n}$ smallest elements in sorted order.
   (a) We can do this in $O(\sqrt{n})$ operations
   (b) We can do this in $O(\sqrt{n} \log n)$ operations
   (c) We can do this in $O(n)$ operations
   (d) We can do this in $O(n \log n)$ operations
   (e) We can do this in $O(n^{1.5})$ operations
3. Define the binary operator @ s.t. $a \mathbin{@} b = \frac{a+b}{2}$. Given an expression $a \mathbin{@} b \mathbin{@} c \mathbin{@} \cdots$, finding the evaluation order that maximizes the total value
   (a) Can be done in $O(n^2)$ time via DP
   (b) Can be done in $O(n^3)$ time via DP
   (c) Can be done in $O(n^4)$ time via DP
   (d) Cannot be done in polynomial time, but can be done in exponential time
   (e) Can be done in $O(n!)$ time, and no faster algorithm is possible

Practice Problems II

4. Given a directed graph G with edges and some vertices s and t, some of which are negative, finding the shortest simple path from s to t using at most 374 negative edges (faster is better)...
   (a) Can be done in $O(m \log n)$ time using Dijkstra's, but only if G has no negative cycles
   (b) Can be done in $O(m \log n)$ time using Dijkstra's, even if G has a negative cycle
   (c) Can be done in $O(mn)$ time using Bellman-Ford, but only if G has no negative cycles
   (d) Can be done in $O(mn)$ time using Bellman-Ford, even if G has a negative cycle

Practice Problems II

4. Given a directed graph G with edges and some vertices s and t, some of which are negative, finding the shortest simple path from s to t using at most 374 negative edges (faster is better)...
   (a) Can be done in $O(m \log n)$ time using Dijkstra's, but only if G has no negative cycles
   (b) Can be done in $O(m \log n)$ time using Dijkstra's, even if G has a negative cycle
   (c) Can be done in $O(mn)$ time using Bellman-Ford, but only if G has no negative cycles
   (d) Can be done in $O(mn)$ time using Bellman-Ford, even if G has a negative cycle
5. Given a connected DAG G with a single sink t and weighted edges, for a given vertex s, if for a path P, $s(P)$ is the sum of all edge values for that path, computing the sum of the $s(P)$ over *all* $s \to t$ paths...
   (a) Can be calculated for all $v \in G$ in $O(n)$ time
   (b) Can be calculated for a *single* $s \in G$ in $O(n)$ time, but requires $O(n^2)$ time for all possible $s \in G$
   (c) Requires $O(n^2)$ time for a single $s \in G$, but can be calculated in $O(n^2)$ time for all possible $s \in G$
   (d) Requires $O(n^2)$ time for a single $s \in G$, and requires $O(n^3)$ time for all $s \in G$
   (e) None of the above are true

Practice Problems III

6. Solve the recurrence $T(n) = T\left(\frac{3n}{4}\right) + T\left(\frac{n}{4}\right) + O(n^2)$, where $T(n) = O(1)$ when $n \leq 374$.
   (a) $O(n)$
   (b) $O(n^2)$
   (c) $O(n^2 \log n)$
   (d) $O(n \log n)$
   (e) $O(n^3)$

Practice Problems III

6. Solve the recurrence $T(n) = T\left(\frac{3n}{4}\right) + T\left(\frac{n}{4}\right) + O(n^2)$, where $T(n) = O(1)$ when $n \leq 374$.
   (a) $O(n)$
   (b) $O(n^2)$
   (c) $O(n^2 \log n)$
   (d) $O(n \log n)$
   (e) $O(n^3)$
7. The problem of determining if a graph G contains a triangle (cycle of length 3) that includes a specific vertex v can be solved in
   (a) $O(m)$
   (b) $O(n^2)$
   (c) $O(nm)$
   (d) $O(n^3)$
   (e) Is NP-Complete

Practice Problems IV

8. Which of the following is true? (If more than one statement is true, pick the strongest).
   (a) A MST will *never* contain the maximum-weight edge in a cycle, and will *always* contain the minimum-weight edge
   (b) A MST *may* contain the maximum-weight edge in a cycle, but will *always* contain the minimum-weight edge
   (c) A MST will *never* contain the maximum-weight edge in a cycle, but *may* contain the minimum-weight edge
   (d) A MST *may* contain the maximum-weight edge in a cycle, and *may* contain the minimum-weight edge
9. Given a graph G whose edges are colored either red or blue,
   (a) We can find a cycle where at least 1/3 of the edges are blue in $O(n + m)$ time
   (b) We can find a cycle where at least 1/3 of the edges are blue in $O(m \log n)$ time
   (c) We can find a cycle where at least 1/3 of the edges are blue in $O(nm)$ time
   (d) We can find a cycle where at least 1/3 of the edges are blue in $O(m^k)$ time for some $k > 2$
   (e) Finding a cycle with 1/3 of the edges being blue is *NP*-hard

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.

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.

Formally, an algorithm C is a certifier for problem X when $s \in X$ if and only if there exists string t such that $C(s, t) = \text{true}$.

- t here is a "certificate."
- We can show X is NP by providing this information, and showing C is polynomial-time and t is polynomial-size (with respect to the size of the input s).

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.

Reductions I: Intuition

- **Intuition:** Problem B is "at least as hard" than 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" than 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

Reductions I: Intuition

- **Intuition:** Problem B is “at least as hard” than 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
- A problem is **NP-hard** if the existence of a polynomial-time algorithm for that problem would imply the existence of a polynomial-time algorithm for any problem in NP. We’ll prove that problems are NP-hard by providing **polynomial-time** reductions from a known NP-hard problem to the problem in question.
  - **NP-complete** problems are NP and NP-hard.

Reductions I: Intuition

- **Intuition:** Problem B is “at least as hard” than 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
- A problem is **NP-hard** if the existence of a polynomial-time algorithm for that problem would imply the existence of a polynomial-time algorithm for any problem in NP. We’ll prove that problems are NP-hard by providing **polynomial-time** reductions from a known NP-hard problem to the problem in question.
  - **NP-complete** problems are NP and NP-hard.
- A problem is **undecidable** if *no* algorithm exists that always completes in the right answer. We’ll prove that problems are undecidable by providing reductions from a known undecidable problem to the problem in question.

Reductions I: Intuition

- **Intuition:** Problem B is "at least as hard" than 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
- A problem is **NP-hard** if the existence of a polynomial-time algorithm for that problem would imply the existence of a polynomial-time algorithm for any problem in NP. We'll prove that problems are NP-hard by providing **polynomial-time** reductions from a known NP-hard problem to the problem in question.
  - **NP-complete** problems are NP and NP-hard.
- A problem is **undecidable** if *no* algorithm exists that always completes in the right answer. We'll prove that problems are undecidable by providing reductions from a known undecidable problem to the problem in question.

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 is NP-hard/undecidable you need to do the following:

Reductions II: Tutorial

- To show a problem is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to. 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 is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to. 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're reducing to using the problem that you're reducing from.

Reductions II: Tutorial

- To show a problem is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to. 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're reducing to using the problem that you're reducing from.
  3. Analyze the runtime for your algorithm, and show it is within your target.

Reductions II: Tutorial

- To show a problem is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to. 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're reducing to using the problem that you're reducing from.
  3. Analyze the runtime for your algorithm, and show it is within your target.
  4. Provide a proof of correctness.

Reductions II: Tutorial

- To show a problem is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to. 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're reducing to using the problem that you're reducing from.
  3. Analyze the runtime for your algorithm, and show it is within your target.
  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 is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to. 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're reducing to using the problem that you're reducing from.
  3. Analyze the runtime for your algorithm, and show it is within your target.
  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

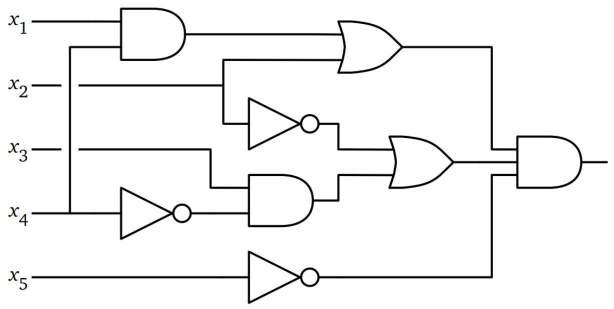
Template- Reduction

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

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

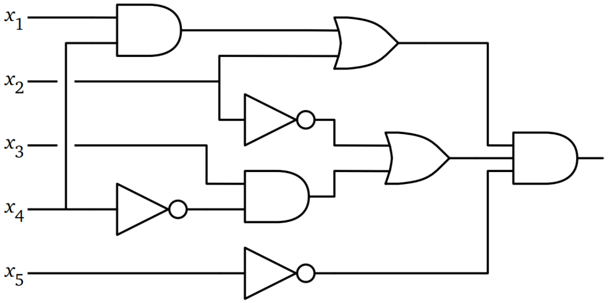
A Tour of NP-Hard Problems: CircuitSAT and 3SAT

- **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

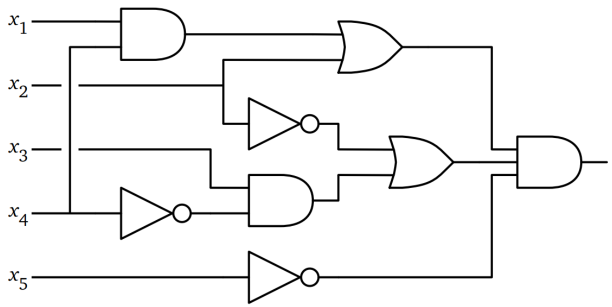
- **CircuitSAT**: The "original" NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



- **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

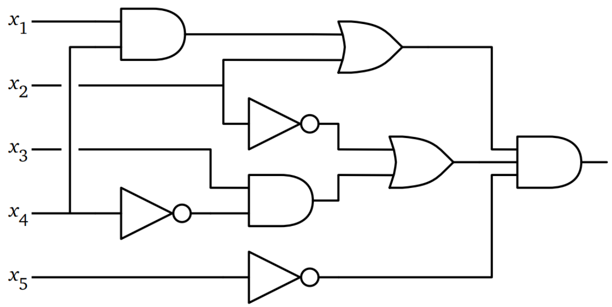
- **CircuitSAT**: The "original" NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



- **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

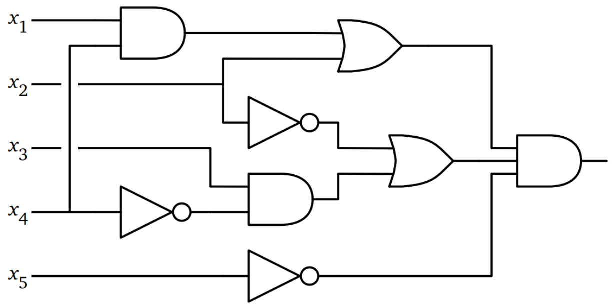
- **CircuitSAT**: The "original" NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



- **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

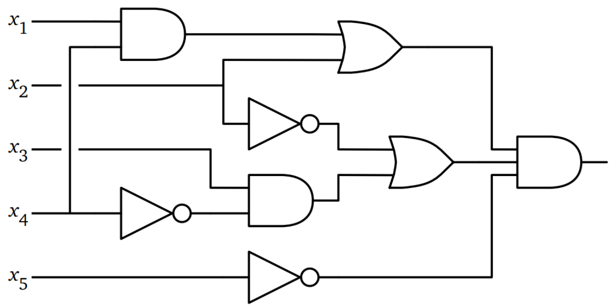
- **CircuitSAT**: The “original” NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



- **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!

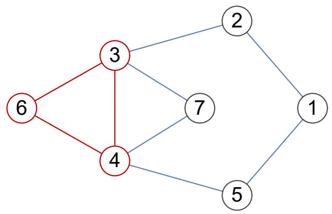
While k-SAT for $k \geq 3$ is NP-complete, there is a polynomial-time algorithm for 2SAT. (Using strongly connected components!)

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

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

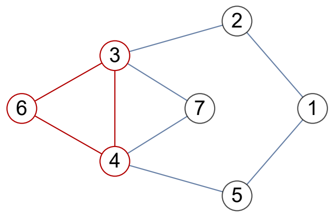
A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

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



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

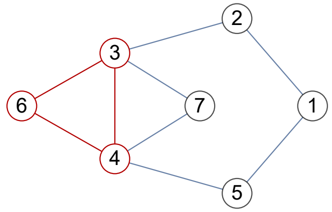
- **MaxClique**: Given a graph G and positive integer h, can we find a K_h subgraph in G (i.e. a set of h nodes where each one has an edge to every other)?



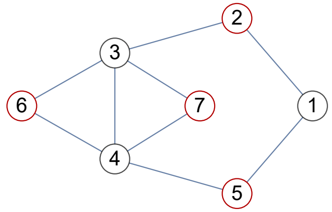
- **MaxIndSet**: Given a graph G and positive integer h, can we find a set of h nodes, none of which share an edge?

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

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

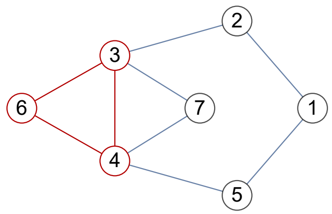


- **MaxIndSet**: Given a graph G and positive integer h, can we find a set of h nodes, none of which share an edge?

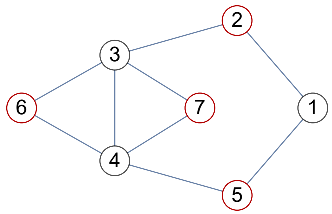


A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

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



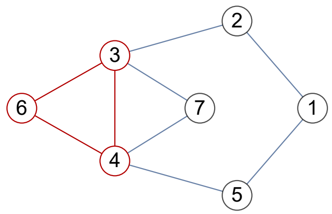
- **MaxIndSet**: Given a graph G and positive integer h, can we find a set of h nodes, none of which share an edge?



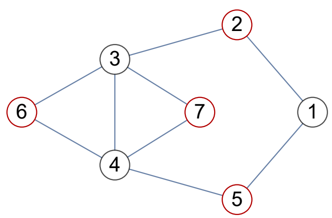
- **MinVertexCover**: Given a graph G and positive integer h, can we find a set of h nodes so that all edges have at least one endpoint chosen?

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

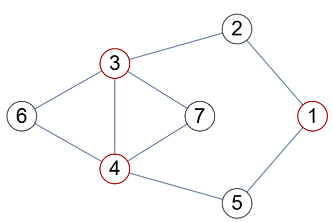
- **MaxClique**: Given a graph G and positive integer h, can we find a K_h subgraph in G (i.e. a set of h nodes where each one has an edge to every other)?



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

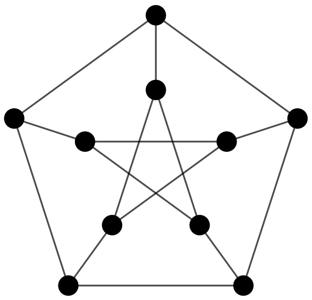


- **MinVertexCover**: Given a graph G and positive integer h, can we find a set of h nodes so that all edges have at least one endpoint chosen?



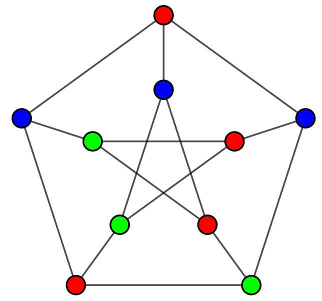
A Tour of NP-Hard Problems: Graph Coloring

- 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?



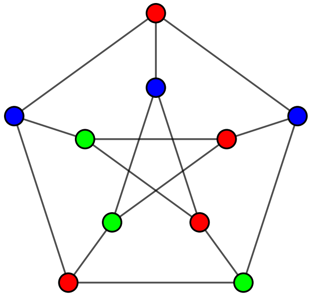
A Tour of NP-Hard Problems: Graph Coloring

- 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

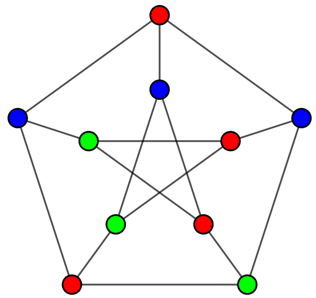
- Given an (undirected) graph, can we color the nodes with at most k colors so that no two vertices that share an edge are of the same color?



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

A Tour of NP-Hard Problems: Graph Coloring

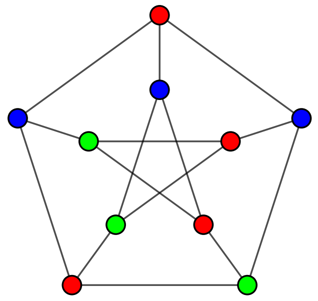
- Given an (undirected) graph, can we color the nodes with at most *k* colors so that no two vertices that share an edge are of the same color?



- 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

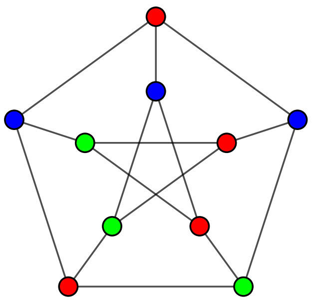
- Given an (undirected) graph, can we color the nodes with at most *k* colors so that no two vertices that share an edge are of the same color?



- 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?



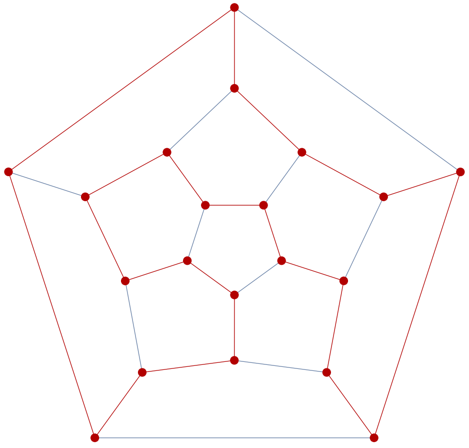
- Consider reducing from k-coloring if. . . :
  - You need to assign objects to groups, and assigning one object to a group limits your choices for some local set of others
  - There's a graph where you need to solve for some *vertex* properties

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.

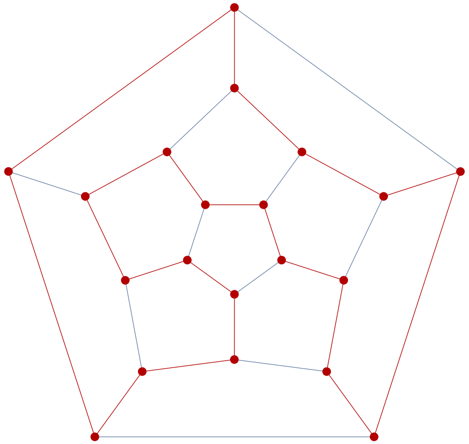
A Tour of NP-Hard Problems: Hamiltonian Paths and Cycles

- 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

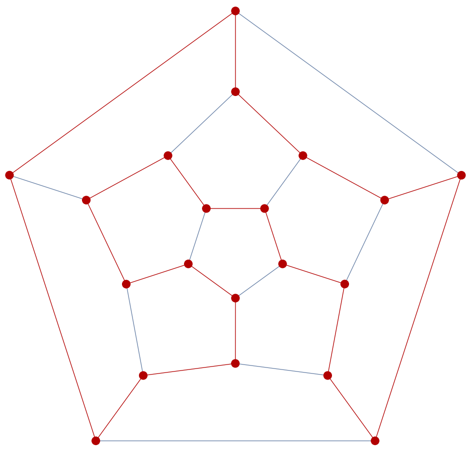
- A **Hamilton Path** is a path that goes through each vertex *exactly* once. Likewise, a **Hamiltonian Cycle** is a cycle that goes through each node *exactly* once.
  - Every graph with a Hamiltonian cycle has a Hamiltonian path, but not every graph with a Hamiltonian path has a Hamiltonian cycle.



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

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

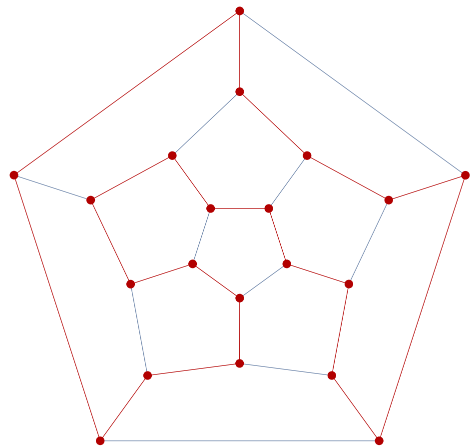
- A **Hamilton Path** is a path that goes through each vertex *exactly* once. Likewise, a **Hamiltonian Cycle** is a cycle that goes through each node *exactly* once.
  - Every graph with a Hamiltonian cycle has a Hamiltonian path, but not every graph with a Hamiltonian path has a Hamiltonian cycle.



- 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 everything

A Tour of NP-hard Problems: Others

- These likely won’t come up on exams, but they’re useful to know.

A Tour of NP-hard Problems: Others

- These likely won't come up on exams, but they're useful to know.
- **LongestPath**: given a (directed, weighted) graph G, is there a path of length at least k?

A Tour of NP-hard Problems: Others

- These likely won't come up on exams, but they're useful to know.
- **LongestPath**: given a (directed, weighted) graph G, is there a path of length at least k?
- **IntegerLinearProgramming**: given a linear objective function to optimize, as well as linear constraints, what is the largest objective achievable where (all/some) variables are restricted to integers?

A Tour of NP-hard Problems: Others

- These likely won't come up on exams, but they're useful to know.
- **LongestPath**: given a (directed, weighted) graph G, is there a path of length at least k?
- **IntegerLinearProgramming**: given a linear objective function to optimize, as well as linear constraints, what is the largest objective achievable where (all/some) variables are restricted to integers?
- **TravelingSalesman**: given a weighted graph G, what is there a Hamiltonian path in G of length at most k?

A Tour of NP-hard Problems: Others

- These likely won't come up on exams, but they're useful to know.
- **LongestPath**: given a (directed, weighted) graph G, is there a path of length at least k?
- **IntegerLinearProgramming**: given a linear objective function to optimize, as well as linear constraints, what is the largest objective achievable where (all/some) variables are restricted to integers?
- **TravelingSalesman**: given a weighted graph G, what is there a Hamiltonian path in G of length at most k?
- **SubsetSum**: given a list of integers, is there a subset that sums to exactly k?

A Tour of NP-hard Problems: Others

- These likely won't come up on exams, but they're useful to know.
- **LongestPath**: given a (directed, weighted) graph G, is there a path of length at least k?
- **IntegerLinearProgramming**: given a linear objective function to optimize, as well as linear constraints, what is the largest objective achievable where (all/some) variables are restricted to integers?
- **TravelingSalesman**: given a weighted graph G, what is there a Hamiltonian path in G of length at most k?
- **SubsetSum**: given a list of integers, is there a subset that sums to exactly k?
- **Checkers**: given a $n \times n$ checkerboard, is there a move that captures at least k checkers?

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`*:* $\{(f, w) : \text{ the function } f \text{ does not infinite loop on input } w\}$ *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`*:* $\{(f, w) : \text{ the function } f \text{ does not infinite loop on input } w\}$ *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:

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.*

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:

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.*

  3. Abuse the fact that you can put code into a function to derive a contradiction.

Practice Problems I

1. Consider the problem of calculating a 3-coloring for a graph G, under the additional constraint that only 374 vertices can be red. Which of the following is true?
   (a) This problem can be solved in $O(m+n)$ time
   (b) This problem can be solved in $O(m^k)$ time for some $k > 1$
   (c) This problem is NP-hard
   (d) This problem is undecidable

Practice Problems I

1. Consider the problem of calculating a 3-coloring for a graph G, under the additional constraint that only 374 vertices can be red. Which of the following is true?
   (a) This problem can be solved in $O(m+n)$ time
   (b) This problem can be solved in $O(m^k)$ time for some $k > 1$
   (c) This problem is NP-hard
   (d) This problem is undecidable
2. Consider the languages
   $\textsc{NeverLeft} = \{\langle M, w\rangle : M \text{ never moves left on input } w\}$ and
   $\textsc{LeftThrice} = \{\langle M, w\rangle : M \text{ moves left on input } w \text{ three times in a row}\}$.
   (a) Neither language is decidable
   (b) Both languages are decidable
   (c) LeftThrice is decidable but not NeverLeft
   (d) NeverLeft is decidable but not LeftThrice

Practice Problems I

1. Consider the problem of calculating a 3-coloring for a graph G, under the additional constraint that only 374 vertices can be red. Which of the following is true?
   (a) This problem can be solved in $O(m+n)$ time
   (b) This problem can be solved in $O(m^k)$ time for some $k > 1$
   (c) This problem is NP-hard
   (d) This problem is undecidable
2. Consider the languages
   $\textsc{NeverLeft} = \{\langle M, w\rangle : M \text{ never moves left on input } w\}$ and
   $\textsc{LeftThrice} = \{\langle M, w\rangle : M \text{ moves left on input } w \text{ three times in a row}\}$.
   (a) Neither language is decidable
   (b) Both languages are decidable
   (c) LEFTTHRICE is decidable but not NEVERLEFT
   (d) NEVERLEFT is decidable but not LEFTTHRICE
3. Consider the problem of detecting Python scripts which could be replaced with DFAs.
   (a) This problem is in P
   (b) This problem is NP-complete
   (c) This problem is decidable
   (d) This problem is undecidable

Practice Problems II

4. Consider the problems LIS, which determines whether the length of the longest increasing subsequence in an array is bigger than some input parameter k, and SingleSourceShortestPath, which determines whether there is a path in a graph G from s to t of length at most ℓ.
   (a) There is a polynomial-time reduction from LIS to SingleSourceShortestPath, but not from SingleSourceShortestPath to LIS
   (b) There is a polynomial-time reduction from SingleSourceShortestPath to LIS, but not from LIS to SingleSourceShortestPath
   (c) There is a polynomial-time reduction from LIS to SingleSourceShortestPath, and from SingleSourceShortestPath to LIS
   (d) There are no polynomial-time reductions between the problems

Practice Problems II

4. Consider the problems LIS, which determines whether the length of the longest increasing subsequence in an array is bigger than some input parameter k, and SINGLESOURCESHORTESTPATH, which determines whether there is a path in a graph G from s to t of length at most ℓ.
   (a) There is a polynomial-time reduction from LIS to SINGLESOURCESHORTESTPATH, but not from SINGLESOURCESHORTESTPATH to LIS
   (b) There is a polynomial-time reduction from SINGLESOURCESHORTESTPATH to LIS, but not from LIS to SINGLESOURCESHORTESTPATH
   (c) There is a polynomial-time reduction from LIS to SINGLESOURCESHORTESTPATH, and from SINGLESOURCESHORTESTPATH to LIS
   (d) There are no polynomial-time reductions between the problems
5. Consider the following problem: Given a list of missions $M_1 \ldots M_n$, each of which consists of a set of intervals indicating the times that you must work on that mission and a number indicating its completion reward, calculate the maximum total reward that you can earn (you can only work on one mission at a time)
   (a) This problem is in P
   (b) This problem is in NP, but it is unknown if it is NP-hard
   (c) This problem is NP-hard but not NP-complete
   (d) This problem is NP-complete

Practice Problems III

6. Consider a weighted directed graph G with n vertices and m edges, where weights can be negative but there are no negative cycles. Finding the longest simple path between two vertices u and v is:
   (a) Possible in $O(n + m)$
   (b) Possible in $O(nm)$ time
   (c) NP-Hard
   (d) Undecidable
   (e) Impossible due to negative weights

Practice Problems III

6. Consider a weighted directed graph G with n vertices and m edges, where weights can be negative but there are no negative cycles. Finding the longest simple path between two vertices u and v is:
   (a) Possible in $O(n + m)$
   (b) Possible in $O(nm)$ time
   (c) NP-Hard
   (d) Undecidable
   (e) Impossible due to negative weights
7. If P = NP, which of the following would be true?
   (a) All decidable problems would be solvable in polynomial time
   (b) Every NP-Hard problem would be solvable in polynomial time
   (c) Every problem in P would be NP-Complete
   (d) The halting problem would become decidable
   (e) Every NP-Complete problem would be solvable in polynomial time

Practice Problems III

6. Consider a weighted directed graph G with n vertices and m edges, where weights can be negative but there are no negative cycles. Finding the longest simple path between two vertices u and v is:
   (a) Possible in $O(n + m)$
   (b) Possible in $O(nm)$ time
   (c) NP-Hard
   (d) Undecidable
   (e) Impossible due to negative weights
7. If P = NP, which of the following would be true?
   (a) All decidable problems would be solvable in polynomial time
   (b) Every NP-Hard problem would be solvable in polynomial time
   (c) Every problem in P would be NP-Complete
   (d) The halting problem would become decidable
   (e) Every NP-Complete problem would be solvable in polynomial time
8. Consider the problem of finding the shortest $s - t$ path in a graph G, except that one edge is a "bridge" which can only used in a specified time interval after the start.
   (a) This problem is in P if you are allowed to wait at vertices, and in P if not
   (b) This problem is NP-hard if you are allowed to wait at vertices, and in P if not
   (c) This problem is in P if you are allowed to wait at vertices, and NP-hard if not
   (d) This problem is NP-hard if you are allowed to wait at vertices, and NP-hard if not

Practice Problems IV

9. A decision problem X is NP-Hard if:
   (a) $X \in NP$
   (b) X cannot be solved in polynomial time
   (c) X reduces to every NP problem in polynomial time
   (d) Every problem in NP reduces to X in polynomial time
   (e) X is undecidable
10. Which of the following problems is decidable?
    (a) Whether a TM accepts 5 strings
    (b) Whether a TM rejects a context-free language
    (c) Whether a TM accepts "UIUC"
    (d) Whether a TM halts on w in $|w|^{374}$ time.

Feedback

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