(b) Prove that kColor problem is NP-hard for any $k \geq 3$.

Solution (direct): The lecture notes include a proof that 3Color is NP-hard. For any integer $k > 3$, I'll describe a direct polynomial-time reduction from 3Color to kColor.

Let G be an arbitrary graph. Let H be the graph obtain from G by adding $k-3$ new vertices $a_1, a_2, \ldots, a_{k-3}$, each with edges to every other vertex in H (including the other a_i's). I claim that G is 3-colorable if and only if H is k-colorable.

$\Longrightarrow$ Suppose G is 3-colorable. Fix an arbitrary 3-coloring of G. Color the new vertices $a_1, a_2, \ldots, a_{k-3}$ with $k-3$ new distinct colors. Every edge in H is either an edge in G or uses at least one new vertex a_i; in either case, the endpoints of the edge have different colors. We conclude that H is k-colorable.

$\Longleftarrow$ Suppose H is k-colorable. Each vertex a_i is adjacent to every other vertex in H, and therefore is the only vertex of its color. Thus, the vertices of G use only three distinct colors. Every edge of G is also an edge of H, so its endpoints have different colors. We conclude that the induced coloring of G is a proper 3-coloring, so G is 3-colorable.

Given G, we can construct H in polynomial time by brute force. ■

2. A *Hamiltonian cycle* in a graph G is a cycle that goes through every vertex of G exactly once. Deciding whether an arbitrary graph contains a Hamiltonian cycle is NP-hard.

A ***tonian cycle*** in a graph G is a cycle that goes through at least *half* of the vertices of G. Prove that deciding whether a graph contains a tonian cycle is NP-hard.

Solution (duplicate the graph): I'll describe a polynomial-time reduction from HamiltonianCycle. Let G be an arbitrary graph. Let H be a graph consisting of two disjoint copies of G, with no edges between them; call these copies G_1 and G_2. I claim that G has a Hamiltonian cycle if and only if H has a tonian cycle.

$\Longrightarrow$ Suppose G has a Hamiltonian cycle C. Let C_1 be the corresponding cycle in G_1. C_1 contains exactly half of the vertices of H, and thus is a tonian cycle in H.

$\Longleftarrow$ On the other hand, suppose H has a tonian cycle C. Because there are no edges between the subgraphs G_1 and G_2, this cycle must lie entirely within one of these two subgraphs. G_1 and G_2 each contain exactly half the vertices of H, so C must also contain exactly half the vertices of H, and thus is a *Hamiltonian* cycle in either G_1 or G_2. But G_1 and G_2 are just copies of G. We conclude that G has a Hamiltonian cycle.

Given G, we can construct H in polynomial time by brute force. ■

Solution (add *n* new vertices): I'll describe a polynomial-time reduction from HamiltonianCycle. Let G be an arbitrary graph, and suppose G has n vertices. Let H be a graph obtained by adding n new vertices to G, but no additional edges. I claim that G has a Hamiltonian cycle if and only if H has a tonian cycle.

$\Longrightarrow$ Suppose G has a Hamiltonian cycle C. Then C visits exactly half the vertices of H, and thus is a tonian cycle in H.

$\Longleftarrow$ On the other hand, suppose H has a tonian cycle C. This cycle cannot visit any of the new vertices, so it must lie entirely within the subgraph G. Since G contains exactly half the vertices of H, the cycle C must visit every vertex of G, and thus is a Hamiltonian cycle in G.

Given G, we can construct H in polynomial time by brute force. ■

A ***clique*** is another name for a complete graph, that is, a graph where every pair of vertices is connected by an edge. The MaxClique problem asks for the number of nodes in its largest complete subgraph in a given graph. A ***vertex cover*** of a graph is a set of vertices that touches every edge in the graph. The MinVertexCover problem is to find the size of the smallest vertex cover in a given graph.

We can prove that MaxClique is NP-hard using the following easy reduction from MaxIndSet. Any graph G has an *edge-complement* $\overline{G}$ with the same vertices, but with exactly the opposite set of edges—(u, v) is an edge in $\overline{G}$ if and only if it is *not* an edge in G. A set of vertices is independent in G if and only if the same vertices define a clique in $\overline{G}$. Thus, the largest independent in G has the same vertices as the largest clique in the complement of G.

The proof that MinVertexCover is NP-hard is even simpler, because it relies on the following easy observation: I is an independent set in a graph $G = (V, E)$ if and only if its complement $V \setminus I$ is a vertex cover of the same graph G. Thus, the *largest* independent set in any graph is just the complement of the *smallest* vertex cover of the same graph! Thus, if the smallest vertex cover in an n-vertex graph has size k, then the largest independent set has size $n - k$.

1. Given an undirected graph G, does G contain a simple path that visits all but 374 vertices?

Solution: We prove this problem is NP-hard by a reduction from the undirected Hamiltonian path problem. Given an arbitrary graph G, let H be the graph obtained from G by adding 374 isolated vertices. Call a path in H ***almost-Hamiltonian*** if it visits all but 374 vertices. I claim that G contains a Hamiltonian path if and only if H contains an almost-Hamiltonian path.

$\Rightarrow$ Suppose G has a Hamiltonian path P. Then P is an almost-Hamiltonian path in H, because it misses only the 374 isolated vertices.

$\Leftarrow$ Suppose H has an almost-Hamiltonian path P. This path must miss all 374 isolated vertices in H, and therefore must visit every vertex in G. Every edge in H, and therefore every edge in P, is also na edge in G. We conclude that P is a Hamiltonian path in G.

Given G, we can easily build H in polynomial time by brute force. ■

3. Prove that the following problem is NP-hard: Given an undirected graph G and an integer k, decide whether the vertices of G can be partitioned into k cliques.

Solution: We prove the problem is NP-hard using a reduction from kColor, which we proved NP-hard in Friday's lab.

Let $G = (V, E)$ be an arbitrary undirected graph. Let $\overline{G} = (V, \overline{E})$ denote the edge-complement of G, where $uv \in \overline{E}$ if and only if $uv \notin E$, for all vertices u and v. I claim that G is k-colorable if and only if the vertices of $\overline{G}$ can be partitioned into k cliques.

$\Longrightarrow$ Suppose G is k-colorable. Fix a proper k-coloring, and let $V_1, V_2, \ldots, V_k$ be the subsets of V of each color. By definition of "proper coloring", for every index i and every pair of indices $u, v \in V_i$, we have $uv \notin E$. Thus, for every index i and every pair of indices $u, v \in V_i$, we have $uv \in \overline{E}$. In other words, each subset V_i is a clique in $\overline{G}$. We conclude that the vertices of $\overline{G}$ can be partitioned into k cliques.

$\Longleftarrow$ Suppose the vertices of $\overline{G}$ can be partitioned into k cliques $V_1, V_2, \ldots, V_k$. By definition of "clique", for every index i and every pair of indices $u, v \in V_i$, we have $uv \in \overline{E}$. Thus, for every index i and every pair of indices $u, v \in V_i$, we have $uv \notin E$. In other words, each subset V_i is an independent set in G; equivalently, if we assign "color" i to each vertex in V_i, for every index i, we obtain a proper coloring of G. We conclude that G is k-colorable.

Solution: This is a simple graph modeling problem. We model the board and allowable move as a directed graph $G = (V, E)$ as follows.

- For each board position (a, b) that is *not* occupied we create a vertex $v_{a,b}$. We assume for simplicity that (i, j) and (i', j') are not occupied by other pieces, otherwise there is no feasible solution for the given problem.
- Suppose $v_{a,b}$ and $v_{a',b'}$ are two vertices that are created in the preceding step. We add an edge $(v_{a,b}, v_{a',b'})$ if a knight can move from (a, b) to (a', b') in one step.

Once the graph G is constructed we run BFS on G to check if there is a path from $v_{i,j}$ to $v_{i',j'}$, and if there is, to find a shortest path. We output "not possible" if there is no path, otherwise we ouput the length of the shortest path.

The running time consists of two parts. First, to create the graph, and second to run BFS on it. The first part takes $O(n^2 + N + M)$ time where where N is number of nodes in G and M is number of edges in G. Running BFS takes $O(N + M)$ time. N can be at most n^2 and M is $O(N)$ since the number of edges out of any node can be at most 8. Thus the total running time is $O(n^2)$. ■

[✓ Yes] No $\varnothing$

To quote the Circle Jerks: "Deny everything! Deny everything!"

[✓ Yes] No $\{ww \mid w \text{ is a palindrome}\}$

Two for-loops.

[✓ Yes] No $\{\langle M \rangle \mid M \text{ is a Turing machine}\}$

This is straightforward syntax-checking.

Yes [✓ No] $\{\langle M \rangle \mid M \text{ accepts } \langle M \rangle \bullet \langle M \rangle\}$

By self-contradiction or reduction from Accept, like Homework 11 problem 1

Yes [✓ No] $\{\langle M \rangle \mid M \text{ accepts an infinite number of palindromes}\}$

Rice's Theorem.

Yes [✓ No] $\{\langle M \rangle \mid M \text{ accepts } \varnothing\}$

Rice's Theorem.

Yes [✓ No] $\{\langle M, w \rangle \mid M \text{ accepts } www\}$

By reduction from Accept. Intuitively, we have no way to distinguish between running forever and just not accepting yet.

Yes [✓ No] $\{\langle M, w \rangle \mid M \text{ accepts } w \text{ after } \textbf{at least } |w|^2 \text{ transitions}\}$

By reduction from Accept. Intuitively, we have no way to distinguish between running forever and just not accepting yet.

Yes [✓ No] $\{\langle M, w \rangle \mid M \text{ changes a non-blank on the tape to a blank, given input } w\}$

By reduction from Accept. Given any Turing machine M, we can define an equivalent Turing machine M' that writes a new symbol § whenever M writes a blank, and then writes § and then a blank in the same tape cell just before it accepts. Then M accepts w if and only if M' writes a blank over a non-blank given input w.

[✓ Yes] No $\{\langle M, w \rangle \mid M \text{ changes a blank on the tape to a non-blank, given input } w\}$

If M never changes a blank to a non-blank, then all non-blank symbols are limited to the input string, so the tape always contains one of $|\Gamma|^{|w|}$ strings. If the head ever moves more than $|Q|$ steps to the right of the input, then M must be stuck in an infinite loop moving to the right. Otherwise, if M runs for more than $|\Gamma|^{|w|}(|Q| + |w|)$ steps, then M must be stuck in an infinite loop, repeating a cycle of configurations.

[✓ Yes] No Given an empty initial tape, M eventually halts.

In fact, M accepts, because $\varepsilon \in$ `0*1*`.

[✓ Yes] No M accepts the string `1111`.

`1111` $\in$ `0*1*`.

Yes [✓ No] M rejects the string `0110`.

`0110` $\notin$ `0*1*`, but M might diverge.

Yes [✓ No] M moves its head to the right at least once, given input `1100`.

M can reject without moving the head as soon as it reads the first `1`.

[✓ Yes] No M moves its head to the right at least once, given input `0101`.

M must read at least the first three symbols; otherwise, it couldn't distinguish between `0101` and `0111`.

[✓ Yes] No M must read a blank before it accepts.

If M accepted some string w without ever reading the first blank after w, then M would also incorrectly accept the string w`010`.

Yes [✓ No] For some input string, M moves its head to the left at least once.

M might simulate a DFA that scans the entire input from left to right, without changing the tape, and then accepts or rejects immedaitely upon reading the first blank.

Yes [✓ No] For some input string, M changes at least one symbol on the tape.

M might simulate a DFA that scans the entire input from left to right, without changing the tape, and then accepts or rejects immedaitely upon reading the first blank.

Yes [✓ No] M always halts.

M might diverge if the input is not in `0*1*`.

Yes [✓ No] If M accepts a string w, it does so after at most $O(|w|^2)$ steps.

After determining that $w \in$ `0*1*`, the machine might wander around doing nothing interesting for $2^{2^{|w|}}$ steps before finally accepting.

For each of the following languages over the alphabet $\Sigma = \{0, 1\}$, either ***prove*** that the language is regular, or ***prove*** that the language is not regular.

(a) $\{www \mid w \in \Sigma^*\}$

Solution: Consider two arbitrary strings $x = 0^i1$ and $y = 0^j1$, where $i \neq j$. Let $z = 0^i10^i1$. Then

- $xz = 0^i10^i10^i1 = (0^i1)^3 \in L$.
- $yz = 0^j10^i10^i1 \notin L$.

Thus, 0^*1 is an infinite fooling set for L, which implies that L is ***not regular***. ■

Rubric: 5 points = 1 for "not regular" + 4 for proof (standard fooling set rubric)

(b) $\{wxw \mid w, x \in \Sigma^*\}$

Solution: For any string $z \in \Sigma^*$, we can write $z = wxw$, where $w = \varepsilon \in \Sigma^*$ and $x = z \in \Sigma^*$. Therefore $\{wxw \mid w, x \in \Sigma^*\} = \Sigma^*$, which is ***regular***. ■

Consider the following pair of languages:

- HamiltonianPath := $\{G \mid G \text{ contains a Hamiltonian path}\}$
- Connected := $\{G \mid G \text{ is connected}\}$

Which of the following ***must*** be true, assuming P≠NP?

[✓ Yes] No Connected $\in$ NP

To verify that a graph is connected in polynomial time, run whatever-first search from any node.

[✓ Yes] No HamiltonianPath $\in$ NP

We can verify that a given path in G is Hamiltonian in polynomial time.

Yes [✓ No] HamiltonianPath is undecidable.

We can decide whether a graph has a Hamiltonian path as follows: For all simple paths π in G, check whether π is Hamiltonian. Yes, this requires exponential time, but that's fine.

Yes [✓ No] There is a polynomial-time reduction from HamiltonianPath to Connected.

This would imply P=NP.

[✓ Yes] No There is a polynomial-time reduction from Connected to HamiltonianPath.

To determine whether a graph G is connected in polynomial time, run whatever-first search from any node. Then, if you absolutely insist, call the magic HamiltonianPath subroutine on a graph with two nodes, which are connected by an edge if and only if G is connected.

Suppose we want to prove that the following language is undecidable.

$$\text{AlwaysHalts} := \{\langle M \rangle \mid M \text{ halts on every input string}\}$$

Bullwinkle J. Moose suggests a reduction from the standard halting language

$$\text{Halt} := \{\langle M, w \rangle \mid M \text{ halts on inputs } w\}.$$

Specifically, suppose there is a Turing machine AH that decides AlwaysHalts. Bullwinkle claims that the following Turing machine H decides Halt. Given an arbitrary encoding $\langle M, w \rangle$ as input, machine H writes the encoding $\langle M' \rangle$ of a new Turing machine M' to the tape and passes it to AH, where M' implements the following algorithm:

```
M'(x):
  if M accepts w
      reject
  if M rejects w
      accept
```

Which of the following statements is true for all inputs $\langle M, w \rangle$?

[✓ Yes] No If M accepts w, then M' halts on every input string.

In fact, M' rejects every string.

[✓ Yes] No If M rejects w, then M' halts on every input string.

In fact, M' accepts every string.

Yes [✓ No] If M rejects w, then H rejects $\langle M, w \rangle$.

M rejects $w \Longrightarrow M'$ accepts every input string $\Longrightarrow M'$ halts on every input string $\Longrightarrow \langle M' \rangle \in$ AlwaysHalts $\Longrightarrow AH$ accepts $\langle M' \rangle \Longrightarrow H$ accepts $\langle M, w \rangle$.

Yes [✓ No] If M diverges on w, then H diverges on $\langle M, w \rangle$.

M diverges $w \Longrightarrow M'$ diverges on every input string $\Longrightarrow$ It is not true that M' halts on every input string $\Longrightarrow \langle M' \rangle \notin$ AlwaysHalts $\Longrightarrow AH$ rejects $\langle M' \rangle \Longrightarrow H$ rejects $\langle M, w \rangle \Longrightarrow H$ halts on $\langle M, w \rangle$.

Yes [✓ No] H does not correctly decide the language Halt. (That is, Bullwinkle's reduction is incorrect.)

Bullwinkle is right!

Let L be any language over a finite alphabet Σ and let $L_{374} = \{w \in L : |w| \geq 374\}$ be the set of strings in L that are of length 374 or more. Prove that if L is regular, then L_{374} is regular.

Solution: There are several ways to solve this:

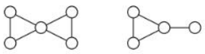
- By closure properties.
  Let $\Sigma^{<374} = \{w : |w| < 374\}$. Then $\Sigma^{<374}$ is finite, hence regular. And, so its complement, $\Sigma^{\geq 374}$ is also regular. Then $L_{374} = L \cap \Sigma^{\geq 374}$ is regular because it is the intersection of two regular languages.
- By describing a machine M_{374} to accept L_{374}.
  Since L is regular, let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA accepting L.
  Basically, we'll add a finite counter to M's states, so that while behaving like an acceptor for L, it additionally counts characters and only accepts if the count has exceeded 373.
  Define $M_{374} = (Q_{374}, \Sigma, \delta_{374}, \text{start}, F_{374})$ as follows.
  - $Q_{374} = Q \times \{0, 1, \ldots, 374\}$
  - $\delta_{374}(\langle q, i \rangle, a) = \langle \delta(q, a), i + 1 \rangle$ if $i \leq 373$, and $\langle \delta(q, a), 374 \rangle$ otherwise.
  - $\text{start} = \langle q_0, 0 \rangle$
  - $F_{374} = \{\langle f, 374 \rangle \mid f \in F\}$

  It should be clear by construction that the machine simulates M in its first component, and keeps track of the character count up to 374 in the second component.

A ***near-Hamiltonian cycle*** in a graph G is a closed walk in G that visits one vertex exactly twice and every other vertex exactly once.

(a) Give an example of a graph that contains a near-Hamiltonian cycle, but does not contain a Hamiltonian cycle (which visits every vertex exactly once).

Solution: Here are two such graphs:



■

Rubric: 2 points

(b) ***Prove*** that it is NP-hard to determine whether a given graph contains a near-Hamiltonian cycle.

Solution: I'll prove the problem is hard by reduction from the usual Hamiltonian cycle problem.

Let G be an arbitrary graph. Let H be the graph obtained from G by adding a new vertex x with a new edge xv to an arbitrary vertex v in G.

- Suppose G contains a Hamiltonian cycle C; this cycle must visit v exactly once. Let C' be the closed walk in H obtained from C by replacing v with $v \to x \to v$. Then C' is a near-Hamiltonian cycle in H.
- Suppose H contains a Hamiltonian cycle C'. This closed walk must visit x, and therefore must visit v twice, and therefore must visit every other vertex of H exactly once. Let C be the closed walk obtained from C' by contracting the subwalk $v \to x \to v$ with v. Then C is a Hamiltonian cycle in G.

We can obviously construct H in polynomial time. ■

String Induction

The ***reversal*** $\boldsymbol{w^R}$ of a string w is defined recursively as follows:

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = ax \text{ for some symbol } a \text{ and some string } x \end{cases}$$

For example, $\texttt{STRESSED}^R = \texttt{DESSERTS}$ and $\texttt{WTF374}^R = \texttt{473FTW}$.

1. Prove that $|w^R| = |w|$ for every string w.

Solution (induction on w):

Let w be an arbitrary string.

Assume for any string x where $|x| < |w|$ that $|x^R| = |x|$.

There are two cases to consider.

- If $w = \varepsilon$, then

$$\begin{aligned} |w^R| &= |\varepsilon| && \text{by definition of }^R \\ &= |w| && \text{by definition of } |\cdot| \end{aligned}$$

- Otherwise, $w = ax$ for some symbol a and some string x. In that case, we have

$$\begin{aligned} |w^R| &= |x^R \bullet a| && \text{by definition of } w^R \\ &= |x^R| + |a| && \text{by Lemma 2} \\ &= |x^R| + 1 && \text{by definition of } |\cdot| \text{ (twice)} \\ &= |x| + 1 && \text{by the induction hypothesis} = |w| \quad \text{by definition of } |\cdot| \end{aligned}$$

In both cases, we conclude that $|w^R| = |w|$.

Product Construction

1. All strings in which the number of $\texttt{0}$s is even and the number of $\texttt{1}$s is *not* divisible by 3.

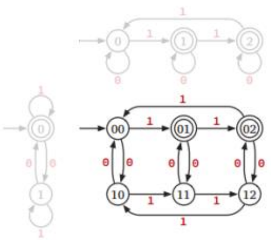
Solution: We use a standard product construction of two DFAs, one accepting strings with an even number of $\texttt{0}$s, and the other accepting strings where the number of $\texttt{1}$s is not a multiple of 3.

The product DFA has six states, each labeled with a pair of integers, one indicating the number $\texttt{0}$s read modulo 2, the other indicating the number of $\texttt{1}$s read modulo 3.

$$\begin{aligned} Q &:= \{0,1\} \times \{0,1,2\} \\ s &:= (0,0) \\ A &:= \{(0,1),(0,2)\} \end{aligned}$$

$$\begin{aligned} \delta((q,r),\texttt{0}) &:= (q+1 \bmod 2,\ r) \\ \delta((q,r),\texttt{1}) &:= (q,\ r+1 \bmod 3) \end{aligned}$$

In this case, the product DFA is simple enough that we can just draw it out in full. I've drawn the two factor DFAs (in gray) to the left and above for reference.



Let G be a connected undirected graph. Suppose we start with two coins on two arbitrarily chosen vertices of G. At every step, each coin *must* move to an adjacent vertex. Describe and analyze an algorithm to compute the minimum number of steps to reach a configuration where both coins are on the same vertex, or to report correctly that no such configuration is reachable. The input o your algorithm consists of a graph $G = (V,E)$ and two vertices $u, v \in V$ (which may or may not e distinct).

olution (product construction): Let $G = (V,E)$ denote the input graph, and let s and t lenote the initial locations of the two coins. We reduce to a shortest-path problem in an ndirected graph $G' = (V',E')$ as follows:

- $V' = V \times V = \{(u,v) \mid u \in V \text{ and } v \in V\}$; the vertices of G' correspond to possible placements of the two coins.
- $E' = \{(u,v)(u',v') \mid uu' \in E \text{ and } vv' \in E\}$. The edges of G' correspond to legal moves by the two coins. Edges are undirected, because any move by the two coins can be reversed.
- We do not need to associate additional values with the vertices or edges.
- We need to find the shortest-path distance from vertex (s,t) to any vertex of the form (v,v).
- First we compute the shortest-path distance from (s,t) to every vertex in G' that is reachable from (s,t) using breadth-first search. Then a simple for-loop over the vertices of the input graph G finds the minimum distance to any marked vertex of the form (v,v). In particular, if no vertex (v,v) is reachable from (s,t), then no vertex (v,v) will be marked by the breadth-first search, and so the algorithm will correct report $\min \varnothing = \infty$.
- The resulting algorithm runs in $O(V' + E') = O(V^2 + E^2)$ time. ■

DFA/NFA Transformation

3. Let L be an arbitrary regular language.

(a) Prove that the language $\mathit{palin}(L) := \{w \mid ww^R \in L\}$ is also regular.

Solution: Let $M = (\Sigma, Q, s, A, \delta)$ be an arbitrary DFA that accepts L. We define a new NFA $M' = (\Sigma, Q', s', A', \delta')$ with ε-transitions that accepts $\mathit{palin}(L)$ as follows:

$$\begin{aligned} Q' &:= (Q \times Q) \cup \{s'\} \\ s' &\text{ is an explicit state in } Q' \\ A' &= \{(q,q) \mid q \in Q\} \\ \delta'(s',\varepsilon) &= \{(s,q) \mid q \in A\} \\ \delta'((p,q),a) &= \{(\delta(p,a),q') \mid \delta(q',a) = q\} \end{aligned}$$

M' reads its input string w and simulates M reading the input string ww^R. Specifically, M' simulates two copies of M, one running forward from the start state s, and the other running backward starting from an accept state.

- The new start state s' non-deterministically guesses the final accept state of M on input ww^R.
- State (p,q) means that the forward (left) copy of M is in state p, and the backward (right) copy of M is in state q.
- M' accepts if and only if the forward simulation of M on w and the backward simulation of M on w^R meet at the same halfway state. ■

4. Let L be an arbitrary regular language. Prove that the language $\mathit{half}(L) := \{w \mid ww \in L\}$ is also regular.

Solution: Let $M = (\Sigma, Q, s, A, \delta)$ be an arbitrary DFA that accepts L. We define a new NFA $M' = (\Sigma, Q', s', A', \delta')$ with ε-transitions that accepts $\mathit{half}(L)$, as follows:

$$\begin{aligned} Q' &= (Q \times Q \times Q) \cup \{s'\} \\ s' &\text{ is an explicit state in } Q' \\ A' &= \{(h,h,q) \mid h \in Q \text{ and } q \in A\} \\ \delta'(s',\varepsilon) &= \{(s,h,h) \mid h \in Q\} \\ \delta'((p,h,q),a) &= \{(\delta(p,a),h,\delta(q,a))\} \end{aligned}$$

M' reads its input string w and simulates M reading the input string ww. Specifically, M' simultaneously simulates two copies of M, one reading the left half of ww starting at the usual start state s, and the other reading the right half of ww starting at some intermediate state h.

- The new start state s' non-deterministically guesses the "halfway" state $h = \delta^*(s,w)$ without reading any input; this is the only non-determinism in M'.
- State (p,h,q) means the following:
  - The left copy of M (which started at state s) is now in state p.
  - The initial guess for the halfway state is h.
  - The right copy of M (which started at state h) is now in state q.
- M' accepts if and only if the left copy of M ends at state h (so the initial non-deterministic guess $h = \delta^*(s,w)$ was correct) and the right copy of M ends in an accepting state. ■

Regular Languages

1. All strings containing the substring $\texttt{000}$.

Solution: $(\texttt{0}+\texttt{1})^*\texttt{000}(\texttt{0}+\texttt{1})^*$

2. All strings *not* containing the substring $\texttt{000}$.

Solution: $(\texttt{1}+\texttt{01}+\texttt{001})^*(\varepsilon+\texttt{0}+\texttt{00})$

3. All strings in which every run of $\texttt{0}$s has length at least 3. **Solution:** $(\texttt{1}+\texttt{0000}^*)^*$

4. All strings in which every substring $\texttt{000}$ appears after every $\texttt{1}$. **Solution:** $(\texttt{1}+\texttt{01}+\texttt{001})^*\texttt{0}^*$

5. All strings containing at least three $\texttt{0}$s. **Solution:** $(\texttt{0}+\texttt{1})^*\texttt{0}(\texttt{0}+\texttt{1})^*\texttt{0}(\texttt{0}+\texttt{1})^*\texttt{0}(\texttt{0}+\texttt{1})^*$

6. Every string except $\texttt{000}$. *[Hint: Don't try to be clever.]*

$$\begin{aligned} &\varepsilon+\texttt{0}+\texttt{1}+\texttt{00}+\texttt{01}+\texttt{10}+\texttt{11} \\ &+\texttt{001}+\texttt{010}+\texttt{011}+\texttt{100}+\texttt{101}+\texttt{110}+\texttt{111} \\ &+(\texttt{1}+\texttt{0})(\texttt{1}+\texttt{0})(\texttt{1}+\texttt{0})(\texttt{1}+\texttt{0})(\texttt{1}+\texttt{0})^* \end{aligned}$$

7. All strings w such that *in every prefix of* w, the number of $\texttt{0}$s and $\texttt{1}$s differ by at most 1.

Solution: Equivalently, strings that alternate between $\texttt{0}$s and $\texttt{1}$s: $(\texttt{01}+\texttt{10})^*(\varepsilon+\texttt{0}+\texttt{1})$

More NFA Transformation

3. Consider the following recursively defined function on strings:

$$\mathit{stutter}(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ aa \bullet \mathit{stutter}(x) & \text{if } w = ax \text{ for some symbol } a \text{ and some string } x \end{cases}$$

Intuitively, $\mathit{stutter}(w)$ doubles every symbol in w. For example:

- $\mathit{stutter}(\texttt{PRESTO}) = \texttt{PPRREESSTTOO}$
- $\mathit{stutter}(\texttt{HOCUS}\diamond\texttt{POCUS}) = \texttt{HHOOCCUUSS}\diamond\diamond\texttt{PPOOCCUUSS}$

Let L be an arbitrary regular language.

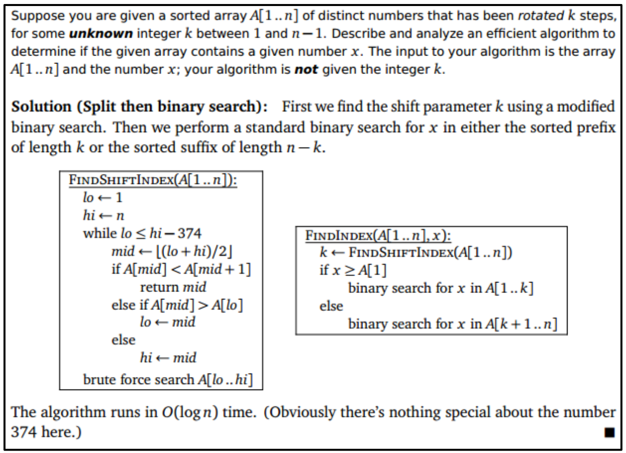
(a) Prove that the language $\mathit{stutter}^{-1}(L) := \{w \mid \mathit{stutter}(w) \in L\}$ is regular.

Solution: Let $M = (\Sigma, Q, s, A, \delta)$ be a DFA that accepts L.

We construct an DFA $M' = (\Sigma, Q', s', A', \delta')$ that accepts $\mathit{stutter}^{-1}(L)$ as follows:

$$\begin{aligned} Q' &= Q \\ s' &= s \\ A' &= A \\ \delta'(q,a) &= \delta(\delta(q,a),a) \end{aligned}$$

M' reads its input string w and simulates M running on $\mathit{stutter}(w)$. Each time M' reads a symbol, the simulation of M reads two copies of that symbol. ■



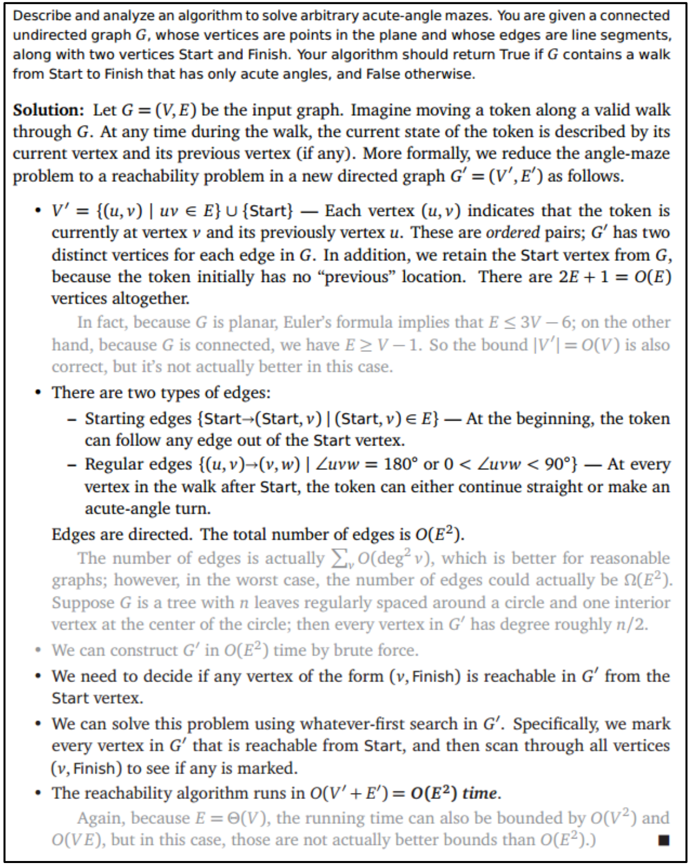
Suppose you are given a sorted array $A[1..n]$ of distinct numbers that has been *rotated* k steps, for some ***unknown*** integer k between 1 and $n-1$. Describe and analyze an efficient algorithm to determine if the given array contains a given number x. The input to your algorithm is the array $A[1..n]$ and the number x; your algorithm is ***not*** given the integer k.

Solution (Split then binary search): First we find the shift parameter k using a modified binary search. Then we perform a standard binary search for x in either the sorted prefix of length k or the sorted suffix of length $n-k$.

```
FindShiftIndex($A[1..n]$):
  $lo \gets 1$
  $hi \gets n$
  while $lo \le hi - 374$
      $mid \gets \lfloor (lo + hi)/2 \rfloor$
      if $A[mid] < A[mid+1]$
          return $mid$
      else if $A[mid] > A[lo]$
          $lo \gets mid$
      else
          $hi \gets mid$
  brute force search $A[lo..hi]$
```

```
FindIndex($A[1..n], x$):
  $k \gets$ FindShiftIndex($A[1..n]$)
  if $x \ge A[1]$
      binary search for $x$ in $A[1..k]$
  else
      binary search for $x$ in $A[k+1..n]$
```

The algorithm runs in $O(\log n)$ time. (Obviously there's nothing special about the number 374 here.) ■



Describe and analyze an algorithm to solve arbitrary acute-angle mazes. You are given a connected undirected graph G, whose vertices are points in the plane and whose edges are line segments, along with two vertices Start and Finish. Your algorithm should return True if G contains a walk from Start to Finish that has only acute angles, and False otherwise.

Solution: Let $G = (V, E)$ be the input graph. Imagine moving a token along a valid walk through G. At any time during the walk, the current state of the token is described by its current vertex and its previous vertex (if any). More formally, we reduce the angle-maze problem to a reachability problem in a new directed graph $G' = (V', E')$ as follows.

- $V' = \{(u, v) \mid uv \in E\} \cup \{\text{Start}\}$ — Each vertex (u, v) indicates that the token is currently at vertex v and its previously vertex u. These are *ordered* pairs; G' has two distinct vertices for each edge in G. In addition, we retain the Start vertex from G, because the token initially has no "previous" location. There are $2E + 1 = O(E)$ vertices altogether.

  In fact, because G is planar, Euler's formula implies that $E \le 3V - 6$; on the other hand, because G is connected, we have $E \ge V - 1$. So the bound $|V'| = O(V)$ is also correct, but it's not actually better in this case.

- There are two types of edges:
  - Starting edges $\{\text{Start} \to (\text{Start}, v) \mid (\text{Start}, v) \in E\}$ — At the beginning, the token can follow any edge out of the Start vertex.
  - Regular edges $\{(u, v) \to (v, w) \mid \angle uvw = 180° \text{ or } 0 < \angle uvw < 90°\}$ — At every vertex in the walk after Start, the token can either continue straight or make an acute-angle turn.

  Edges are directed. The total number of edges is $O(E^2)$.

  The number of edges is actually $\sum_v O(\deg^2 v)$, which is better for reasonable graphs; however, in the worst case, the number of edges could actually be $\Omega(E^2)$. Suppose G is a tree with n leaves regularly spaced around a circle and one interior vertex at the center of the circle; then every vertex in G' has degree roughly $n/2$.

- We can construct G' in $O(E^2)$ time by brute force.
- We need to decide if any vertex of the form (v, Finish) is reachable in G' from the Start vertex.
- We can solve this problem using whatever-first search in G'. Specifically, we mark every vertex in G' that is reachable from Start, and then scan through all vertices (v, Finish) to see if any is marked.
- The reachability algorithm runs in $O(V' + E') = \boldsymbol{O(E^2)}$ ***time***.

  Again, because $E = \Theta(V)$, the running time can also be bounded by $O(V^2)$ and $O(VE)$, but in this case, those are not actually better bounds than $O(E^2)$.) ■

AcceptIllini := $\{\langle M\rangle \mid M$ accepts the string ILLINI$\}$

Solution: For the sake of argument, suppose there is an algorithm DecideAcceptIllini that correctly decides the language AcceptIllini. Then we can solve the halting problem as follows:

```
DecideHalt(⟨M, w⟩):
  Encode the following Turing machine M′:
      M′(x):
        run M on input w
        return True
  if DecideAcceptIllini(⟨M′⟩)
      return True
  else
      return False
```

We prove this reduction correct as follows:

$\Longrightarrow$ Suppose M halts on input w.
Then M' accepts *every* input string x.
In particular, M' accepts the string ILLINI.
So DecideAcceptIllini accepts the encoding $\langle M'\rangle$.
So DecideHalt correctly accepts the encoding $\langle M, w\rangle$.

$\Longleftarrow$ Suppose M does not halt on input w.
Then M' diverges on *every* input string x.
In particular, M' does not accept the string ILLINI.
So DecideAcceptIllini rejects the encoding $\langle M'\rangle$.
So DecideHalt correctly rejects the encoding $\langle M, w\rangle$.

In both cases, DecideHalt is correct. But that's impossible, because Halt is undecidable. We conclude that the algorithm DecideAcceptIllini does not exist. ■

Theorem 19. *The language* NeverLeft $:= \{\langle M, w\rangle \mid$ *Given w as input, M never moves left*$\}$ *is decidable.*

Proof: Given the encoding $\langle M, w\rangle$, we simulate M with input w using our universal Turing machine U, but with the following termination conditions. If M ever moves its head to the left, then we reject. If M halts without moving its head to the left, then we accept. Finally, if M reads more than $|Q|$ blanks, where Q is the state set of M, then we accept. If the first two cases do not apply, M only moves to the right; moreover, after reading the entire input string, M only reads blanks. Thus, after reading $|Q|$ blanks, it must repeat some state, and therefore loop forever without moving to the left. The three cases are exhaustive. □

Theorem 20. *The language* LeftThree $:= \{\langle M, w\rangle \mid$ *Given w as input, M eventually moves left three times in a row*$\}$ *is undecidable.*

Proof: Given $\langle M\rangle$, we build a new Turing machine M' that accepts the same language as M and moves left three times in a row if and only if it accepts, as follows. For each non-accepting state p of M, the new machine M' has three states p_1, p_2, p_3, with the following transitions:

$$\begin{aligned}
\delta'(p_1, a) &= (q_2, b, \Delta), && \text{where } (q, b, \Delta) = \delta(p, a) \text{ and } q \neq \text{accept}\\
\delta'(p_2, a) &= (p_3, a, +1)\\
\delta'(p_3, a) &= (p_1, a, -1)
\end{aligned}$$

In other words, after each non-accepting transition, M' moves once to the right and then once to the left. For each transition to accept, M' has a sequence of seven transitions: three steps to the right, then three steps to the left, and then finally accept', all without modifying the tape. (The three steps to the right ensure that M' does not fall off the left end of the tape.)

Finally, M' moves left three times in a row if and only if M accepts w. Thus, if we could decide LeftThree, we could also decide Accept, which is impossible. □

For any language L, let Suffixes$(L) := \{x \mid yx \in L$ for some $y \in \Sigma^*\}$ be the language containing all suffixes of all strings in L. For example, if $L = \{000, 100, 110, 111\}$, then Suffixes$(L) = \{\varepsilon, 0, 1, 00, 10, 11, 000, 100, 110, 111\}$.

Prove that for any regular language L, the language Suffixes(L) is also regular.

Solution (one new state): Let $M = (\Sigma, Q, s, A, \delta)$ be an arbitrary DFA that recognizes L.

Without loss of generality, we assume that every state in Q is reachable from the start state s; that is, for every $q \in Q$, there is some string $w \in \Sigma^*$ such that $\delta(s, w) = q$. Otherwise, we simply discard any unreachable states to get a smaller DFA that still recognizes L.

We define an NFA $M' = (\Sigma, Q', s', A', \delta')$ as follows:

$$\begin{aligned}
Q' &= Q \cup \{s'\}\\
s' &\text{ is a new explicit state}\\
A' &= A
\end{aligned}$$

$$\begin{aligned}
\delta'(s', \varepsilon) &= Q\\
\delta'(s', a) &= \varnothing && \text{for all } a \in \Sigma\\
\delta'(q, \varepsilon) &= \varnothing && \text{for all } q \in Q\\
\delta'(q, a) &= \{\delta(q, a)\} && \text{for all } q \in Q \text{ and } a \in \Sigma
\end{aligned}$$

In other words, we add a new start state s' with ε-transitions to *every* other state. ■

A *decision problem* is a problem whose output is a single boolean value: Yes or No. Let me define three classes of decision problems:

- ***P*** is the set of decision problems that can be solved in polynomial time. Intuitively, P is the set of problems that can be solved quickly.
- ***NP*** is the set of decision problems with the following property: If the answer is Yes, then there is a *proof* of this fact that can be checked in polynomial time. Intuitively, NP is the set of decision problems where we can verify a Yes answer quickly if we have the solution in front of us.
- ***co-NP*** is essentially the opposite of NP. If the answer to a problem in co-NP is No, then there is a proof of this fact that can be checked in polynomial time.

Lemma 3.8. *The language $L = \{0^{2^n} \mid n \geq 0\}$ is not regular.*

Proof: Let x and y be arbitrary distinct strings in L. Then we must have $x = 0^{2^i}$ and $y = 0^{2^j}$ for some integers $i \neq j$. The suffix $z = 0^{2^i}$ distinguishes x and y, because $xz = 0^{2^i+2^i} = 0^{2^{i+1}} \in L$, but $yz = 0^{2^i+2^j} \notin L$. We conclude that L itself is a fooling set for L. Because L is infinite, L cannot be regular. □

Rice's Theorem. *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$(N) \notin \mathcal{L}$.

The language AcceptIn$(\mathcal{L}) := \{\langle M\rangle \mid$ Accept$(M) \in \mathcal{L}\}$ *is undecidable.*

Corollary 15. *Each of the following languages is undecidable.*
(a) $\{\langle M\rangle \mid M$ *accepts given an empty initial tape*$\}$
(b) $\{\langle M\rangle \mid M$ *accepts the string* UIUC$\}$
(c) $\{\langle M\rangle \mid M$ *accepts exactly three strings*$\}$
(d) $\{\langle M\rangle \mid M$ *accepts all palindromes*$\}$
(e) $\{\langle M\rangle \mid$ Accept(M) *is regular*$\}$
(f) $\{\langle M\rangle \mid$ Accept(M) *is not regular*$\}$
(g) $\{\langle M\rangle \mid$ Accept(M) *is undecidable*$\}$
(h) $\{\langle M\rangle \mid$ Accept$(M) =$ Accept$(N)\}$, *for some arbitrary fixed Turing machine N.*

DFS	O(V+E)	
BFS	O(V+E)	
Dijkstra	O(ElogV)	
Merge Sort	2T(n/2) + O(n)	O(n log n)
Quick Sort	T(r-1) + T(n-r) + O(n)	O(n log n)
Binary Search	T(n/2) + O(n)	O(log n)
Karatsuba's	3T(n/2) + O(n)	Θ(n ^ log 3)
Linear Search	T((n+1)/2) + O(n)	O(n)
Fibonacci	T(n-1) + T(n-2) + O(n)	O(1.618)

For each of the problems below, transform the input into a graph and apply a standard graph algorithm that you've seen in class. Whenever you use a standard graph algorithm, you ***must*** provide the following information. (I recommend actually using a bulleted list.)

- What are the vertices?
- What are the edges? Are they directed or undirected?
- If the vertices and/or edges have associated values, what are they?
- What problem do you need to solve on this graph?
- What standard algorithm are you using to solve that problem?
- What is the running time of your entire algorithm, *including* the time to build the graph, *as a function of the original input parameters*?

Dynamic Programming

Now that we have a recurrence, we can transform it into a dynamic programming algorithm following the usual mechanical boilerplate.

- **Subproblems:** Each recursive subproblem is identified by two indices $0 \leq i \leq m$ and $0 \leq j \leq n$.
- **Memoization structure:** So we can memoize all possible values of $Edit(i, j)$ in a two-dimensional array $Edit[0..m, 0..n]$.
- **Dependencies:** Each entry $Edit[i, j]$ depends only on its three neighboring entries $Edit[i-1, j]$, $Edit[i, j-1]$, and $Edit[i-1, j-1]$.
- **Evaluation order:** So if we fill in our table in the standard row-major order—row by row from top down, each row from left to right—then whenever we reach an entry in the table, the entries it depends on are already available.
- **Space and time:** The memoization structure uses ***O(mn) space***, and we can compute each entry $Edit[i, j]$ in $O(1)$ time one we know its predecessors, so the overall algorithm runs in ***O(mn) time.***

Describe and analyze an algorithm to solve arbitrary acute-angle mazes. You are given a connected undirected graph G, whose vertices are points in the plane and whose edges are line segments, along with two vertices Start and Finish. Your algorithm should return True if G contains a walk from Start to Finish that has only acute angles, and False otherwise.

Solution: Let $G = (V, E)$ be the input graph. Imagine moving a token along a valid walk through G. At any time during the walk, the current state of the token is described by its current vertex and its previous vertex (if any). More formally, we reduce the angle-maze problem to a reachability problem in a new directed graph $G' = (V', E')$ as follows.

- $V' = \{(u, v) \mid uv \in E\} \cup \{\text{Start}\}$ — Each vertex (u, v) indicates that the token is currently at vertex v and its previously vertex u. These are *ordered* pairs; G' has two distinct vertices for each edge in G. In addition, we retain the Start vertex from G, because the token initially has no "previous" location. There are $2E + 1 = O(E)$ vertices altogether.

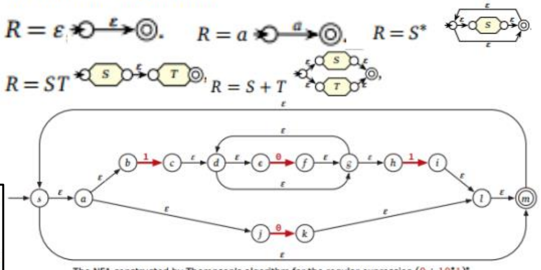
  In fact, because G is planar, Euler's formula implies that $E \leq 3V - 6$; on the other hand, because G is connected, we have $E \geq V - 1$. So the bound $|V'| = O(V)$ is also correct, but it's not actually better in this case.
- There are two types of edges:
  - Starting edges $\{\text{Start} \to (\text{Start}, v) \mid (\text{Start}, v) \in E\}$ — At the beginning, the token can follow any edge out of the Start vertex.
  - Regular edges $\{(u, v) \to (v, w) \mid \angle uvw = 180° \text{ or } 0 < \angle uvw < 90°\}$ — At every vertex in the walk after Start, the token can either continue straight or make an acute-angle turn.

  Edges are directed. The total number of edges is $O(E^2)$.

  The number of edges is actually $\sum_v O(\deg^2 v)$, which is better for reasonable graphs; however, in the worst case, the number of edges could actually be $\Omega(E^2)$. Suppose G is a tree with n leaves regularly spaced around a circle and one interior vertex at the center of the circle; then every vertex in G' has degree roughly $n/2$.
- We can construct G' in $O(E^2)$ time by brute force.
- We need to decide if any vertex of the form (v, Finish) is reachable in G' from the Start vertex.
- We can solve this problem using whatever-first search in G'. Specifically, we mark every vertex in G' that is reachable from Start, and then scan through all vertices (v, Finish) to see if any is marked.
- The reachability algorithm runs in $O(V' + E') = \boldsymbol{O(E^2)}$ ***time***.

  Again, because $E = \Theta(V)$, the running time can also be bounded by $O(V^2)$ and $O(VE)$, but in this case, those are not actually better bounds than $O(E^2)$.) ■

Thompson's Algorithm



The NFA constructed by Thompson's algorithm for the regular expression $(0 + 10^*1)^*$.
The four non-ε-transitions are drawn with with bold red arrows for emphasis.

3. Suppose L is a NP-Complete problem. For each of the following circle "True" if the statement is necessarily true and "False" if there are cases where the statement is false.

   (a) [✓ True] False L is in NP
   (b) [✓ True] False L is NP-Hard
   (c) [✓ True] False there is a polynomial time reduction from every NP-Complete language L' to L
   (d) True [✓ False] there is a polynomial time reduction from every NP-Hard language L' to L
   (e) [✓ True] False there is a polynomial-time reduction from L to **3SAT**.

String induction:
IH: assume **True** for all strings x s.t. |x| < |w|
W = empty string OR ax, where x is a string
Boilerplate: w is arbitrary string, then IH, then if w = empty string, then w = ax, QED.

Regular Languages:
Is either empty set, single string, union or concatenation of two regular languages, or Kleene closure of regular language.
Template: let R be arbitrary regular expression, assume true for every proper subexpression.
Suppose R = empty, single string, S+T, S.T, S*.
Can also induct on size of regular expression; assume true for every expression smaller than R.

If proving language is regular, try to build NFA for it

Graphs:
TOP SORT IS ON DAG ONLY. DFS processes things in reverse-top-sort order
KosarajuSharir gets SCC(G) (strongly conn. Comp) in O(V+E) time
DRAW GRAPH OUT FOR REDUCTION.
Bellman Ford: SSSP in O(VE), relaxes tense edges

Decidability:
Reduction: Reduce X to Y by assuming program Py that decides Y, then use that to decide X.