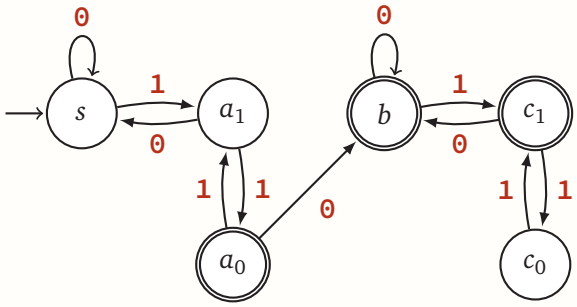
1. (a) Draw an NFA that accepts the language $\{w \mid$ there is exactly one block of 1s of even length$\}$. (A "block of 1s" is a maximal substring of 1s.)

Solution: An NFA for the language is shown below.



To see the correctness of this construction, we describe the meaning of each state.

- s: Haven't seen even block; reading block of 0s.
- a_1: Haven't seen even block; reading block of 1s, current parity is 1.
- a_0: Haven't seen even block; reading block of 1s, current parity is 0.
- b: Seen even block; reading block of 0s.
- c_1: Seen even block; reading block of 1s, current parity is 1.
- c_0: Seen even block; reading block of 1s, current parity is 0.

We now give a summary of this construction. Intuitively, our goal in designing our NFA is, for each block of 1s, to check if this block is a block of even length. After seeing one even length block, we want to make sure than no other blocks have even length. If the number of even length blocks is not equal to 1, we reject.

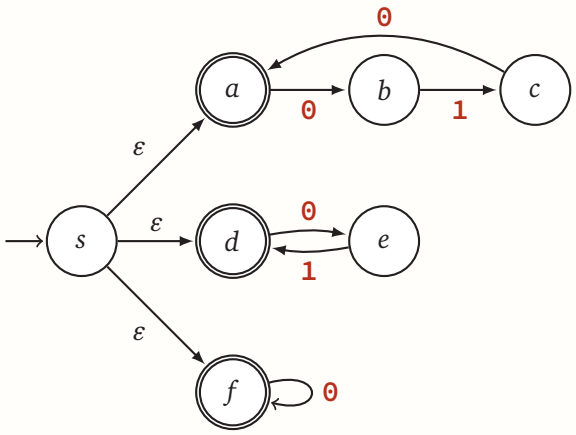
Observe that the initial three states s, a_1, a_0 form a gadget that ignores blocks of 0s and blocks of 1s of odd length. Once a 0 is seen concluding a block of 1s of even length (representing the 0 transition from a_0 to b), then the NFA transitions to another copy of the same gadget.

In this new gadget (with states b, c_1, c_0), there is now no arrow for 0 coming out of the state c_0. This is because, when at state c_0, reading in a 0 (or reaching the end of the input) would represent seeing another block of 1s of even length, meaning that we should reject the input string.

The accepting states are a_0, b, and c_1, since a string concluding at one of these states represents having seen exactly one block of 1s of even length. Specifically, a_0 represents the string ending with the lone block of 1s of even length, b represents the string ending with a block of 0s, and c_1 represents the string ending with a block of 1s of odd length. ■

(b) i. Draw an NFA for the regular expression $(\mathtt{010})^* + (\mathtt{01})^* + \mathtt{0}^*$.

Solution: An NFA corresponding to the regular expression is shown below.



The NFA was created as follows. First, NFAs for the three expressions $(\mathtt{010})^*$, $(\mathtt{01})^*$ and $\mathtt{0}^*$ are drawn separately. These NFAs were then combined by adding a new start state, connected to the start states of the original three NFAs via ε-transitions.

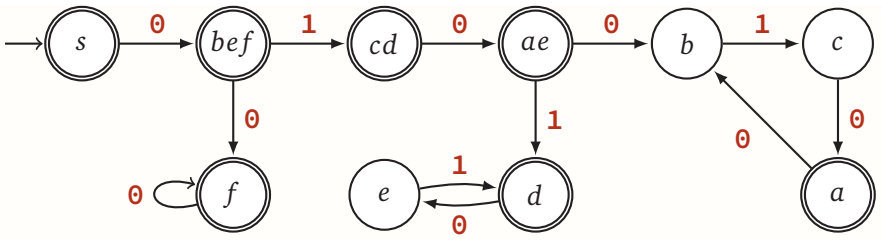
In principal, one could follow Thompson's algorithm directly to the letter to obtain an NFA with 16 states. However, Thompson's algorithm produces many extra states in order to (i) work for arbitrary NFAs, and (ii) enforce useful invariants that simplify the algorithm (but not the construction). ■

ii. Now using the powerset construction (also called the subset construcion), design a DFA for the same language. Label the states of your DFA with names that are sets of states of your NFA. You should use the incremental construction so that you only generate the states that are reachable form the start state.

Solution: We use the incremental power set construction method to construct the equivalent DFA from the NFA of the previous part.

q'	ε-reach	$\mathtt{0}$	$\mathtt{1}$	A'?
s	$sadf$	bef	$\emptyset$	✓
bef	bef	f	cd	✓
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	
f	f	f	$\emptyset$	✓
cd	cd	ae	$\emptyset$	✓
ae	ae	b	d	✓
b	b	$\emptyset$	c	
d	d	e	$\emptyset$	✓
c	c	a	$\emptyset$	
e	e	$\emptyset$	d	
a	a	b	$\emptyset$	✓

The resultant DFA is shown below; *all unspecified transitions lead to the (undrawn) rejecting $\emptyset$ state*.



Both the table and the drawing fully specify the DFA, so just writing down one or the other is sufficient for full credit. ■

Rubric: 10 points.

(a) 5 points
-5 for a completely incorrect NFA.
-1 for each small error (up to two transitions labeled incorrectly, up to one edge missing, adding or deleting one state makes the construction correct)
-1.5 for missing justification.

(b.i) 2 points
-2 for a completely incorrect NFA
-1 for each small errors (transitions labeled incorrectly (e.g. **1** when it should be a **0**). If more than two transitions are mislabeled, consider NFA is completely incorrect.
-0.5 for missing justification, if NFA is simplified. (No justification necessary if Thompson's algorithm output drawn in full)

(b.ii) 3 points
-3 for a completely incorrect DFA
-1 for each small errors (less than three transitions labeled incorrectly, at most one state missing, at most two accept states not labeled as accept states)

2. For a language L let SUFFIX$(L) = \{y \mid \exists x \in \Sigma^*, xy \in L\}$ be the set of suffixes of strings in L. Let PSUFFIX$(L) = \{y \mid \exists x \in \Sigma^*, |x| \geq 1, xy \in L\}$ be the set of proper suffixes of strings in L.

   (a) Prove that if L is regular then PSUFFIX(L) is regular via the following technique. Let $M = (Q, \Sigma, \delta, s, A)$ be a DFA accepting L. Describe a NFA N in terms of M that accepts PSUFFIX(L). Explain the construction of your NFA.

Solution: We will assume that all states of M are reachable from the start state s. Otherwise we can modify M to remove the states not reachable from s and this will not affect the behaviour of M.

Let $X \subseteq Q$ be the set of states of M that are reachable from s on a non-empty string. That is $X = \{q \in Q \mid \delta^*(s, w) = q, |w| \geq 1\}$. Every state is reachable from s and M is a DFA so if $q \neq s$ then $q \in X$ (the only way to reach a state $q \neq s$ is via a non-empty string. If $q = s$ we then we need to check whether there is a non-empty string w such that $\delta^*(s, w) = s$. There is such a string if there is a state $q \neq s$ that can reach s, or if s has a self-loop on some symbol a. In any case we can determine X and $X = Q$ or $X = Q \setminus \{s\}$.

We now define an NFA $N = (Q', \Sigma, \delta', s', A')$ from the DFA $M = (Q, \Sigma, \delta, s, A)$ such that $L(N) = \text{PSUFFIX}(L(M))$, as follows. The NFA N has one additional state t which becomes its start state.

$$\begin{aligned} Q' &:= Q \cup \{t\} \\ \delta'(q, a) &:= \{\delta(q, a)\} \quad \text{for all } q \in Q \text{ and } a \in \Sigma \\ \delta'(t, \varepsilon) &:= \left\{q \in Q \;\middle|\; \delta^*(s, w) = q, |w| \geq 1\right\} \\ s' &:= t \\ A' &:= A \end{aligned}$$

The NFA N starts in t, ***guesses*** a state $q \in X$ (recall X is the set of states reachable from s on a non-empty string) and then simulates M starting from q.

- Suppose N accepts a string w. Since $t \notin A'$, N can accept w only by jumping to some $q \in X$ and then reaching an an accept state of M by simulating M on w from q. This implies that there is a a state $q \in X$ such that $\delta^*_M(q, w) \in A$. Since $q \in X$ there is a non-empty string $u \in \Sigma^*$ such that $\delta^*_M(s, u) = q$. Hence $\delta^*_M(s, uw) \in A$ which implies that $uw \in L(M)$. Therefore $w \in \text{PSUFFIX}(L(M))$.
- Conversely if $w \in \text{PSUFFIX}(L(M))$ then there is a non-empty string $u \in \Sigma^*$ such that $uw \in L(M)$. Let $q = \delta^*_M(s, u)$. We have $q \in X$. Hence there is an accepting path for NFA N on w where N jumps from t to q and then simulates M on w. Thus $w \in L(N)$.

This shows that $L(N) = \text{PSUFFIX}(L(M))$.

The argument that $L(N) = \text{PSUFFIX}(L(M))$ is *not* necessary for full credit. ■

(b) Prove that if L is regular then PSUFFIX(L) is regular via the following alternate technique. Let r be a regular expression. We will develop an algorithm that given r constructs a regular expression r' such that $L(r') = \text{PSUFFIX}(L(r))$. Assume $\Sigma = \{\mathtt{0}, \mathtt{1}\}$. No correctness proof is required but a brief explanation of the derivation would help you get partial credit in case of mistakes.

i. For each of the base cases of regular expressions $\emptyset, \epsilon$ and a, $a \in \Sigma$ describe a regular expression for PSUFFIX($L(r)$).

Solution:
- If $r = \emptyset$, then $L(r) = \emptyset$, and so PSUFFIX($L(r)$) $= \emptyset$ as well. So $r' = \emptyset$.
- If $r = \varepsilon$ then $L(r) = \{\varepsilon\}$. ε cannot be written as xy where $|x| \geq 1$, so PSUFFIX($L(r)$) $= \emptyset$. Once again $r' = \emptyset$.
- On the other hand, when $r = a$, $L(r) = \{a\}$. The only way to write a as xy where $|x| \geq 1$ is to set $x = a$ and $y = \varepsilon$. So PSUFFIX($L(r)$) $= \{\varepsilon\}$, and thus $r' = \varepsilon$. ■

ii. Suppose r_1 and r_2 are regular expressions, and r_1' and r_2' are regular expressions for the languages PSUFFIX($L(r_1)$) and PSUFFIX($L(r_2)$) respectively. Describe a regular expression for the language PSUFFIX($L(r_1 + r_2)$) using r_1, r_2, r_1', r_2'.

Solution: Consider $w \in \text{PSUFFIX}(L(r_1 + r_2)) = \text{PSUFFIX}(L(r_1) \cup L(r_2))$. Then $w = xy$ where $|x| \geq 1$ and $xy \in L(r_1)$ ***or*** $xy \in L(r_2)$. Distributing over the *or*, we can rewrite this as $w \in \text{PSUFFIX}(L(r_1)) \cup \text{PSUFFIX}(L(r_2))$. We conclude that $r' = r_1' + r_2'$. ■

iii. Same as the previous part but now consider $L(r_1 r_2)$.

Solution: Consider w in PSUFFIX($L(r_1 r_2)$). There are two possibilities.
- $w \in \text{PSUFFIX}(L(r_2))$, or
- $w = xy$ where $x \in \text{PSUFFIX}(L(r_1))$ and $y \in L(r_2)$.

The regular expression for the first case is simply r_2'; the regular expression for the second case is $r_1' r_2$. Combining the two gives us $r' = r_2' + r_1' r_2$. ■

iv. Same as the previous part but now consider $L((r_1)^*)$.

Solution: Consider an arbitrary string w in PSUFFIX($L(r_1^*)$). It must begin with a proper suffix of some string in $L(r_1)$, followed by an arbitrary number of strings in $L(r_1)$. In other words, w must be of the form $w = xy$, where $x \in \text{PSUFFIX}(L(r_1)), y \in L(r_1^*)$. This leads us to $r = r_1' r_1^*$. ■

v. Apply your construction to the regular expression $r = 0^* + (01)^* + 011^*0$ to obtain a regular expression for the language PSUFFIX($L(r)$).

Solution: We can apply the previous rules to the expression mechanically and simplify the resulting regular expression to obtain the following:

$$r' = \mathtt{0}^* + (\epsilon + \mathtt{1})(\mathtt{01})^* + (\epsilon + \mathtt{1})(\mathtt{1}^*\mathtt{0} + \mathtt{0}) = \mathtt{0}^* + (\epsilon + \mathtt{1})(\mathtt{01})^* + \mathtt{1}^*\mathtt{0}.$$

Simplifying your answer is *not* necessary for full credit. ■

Rubric: 10 points.

(a) 5 points

+ 3 for correctly constructing N.
  - -0.5 for missing Q, δ, s, A
  - -0.5 for incorrect Q.
  - -0.5 for incorrect δ.
  - -0.5 for incorrect s.
  - -0.5 for incorrect A.
  - Note: Judge correctness based on if NFA N is does accept PSUFFIX(L) for M accepting L.
  - -0.5 for not creating the described NFA AND not creating a NFA that satisfies PSUFFIX(L).
  - Note: This means we give them this .5 if their NFA matches their description, even if the description is wrong OR if their description is wrong, but the NFA is correct.
+ 2 for describing how to construct N.
  - -1 for missing description, but NFA is flawless.
  - -2 for missing description and flawed NFA.
  - -1 for incorrect description.

(b) 5 points

- -0.5 for incorrect part (i)
- -0.5 for incorrect part (ii).
- -1.5 for incorrect part (iii).
- -1.5 for incorrect part (iv).
- -1 for incorrect part (v).

3. Recall that if M is a DFA that accepts a language L then it is easy to construct a DFA M' to accept the language $\overline{L}$ (the complement of L) by simply altering the final states. The product construction allows one to take two DFAs M_1 and M_2 and construct a machine M that accepts $L(M_1) \cap L(M_2)$. Here we explore NFAs. If N_1 and N_2 are two NFAs then we saw in lecture that it is easy to construct a NFA N that accepts $L(N_1) \cup L(N_2)$.

   - Let $N_1 = (Q_1, \Sigma, \delta_1, s_1, A_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, s_2, A_2)$ be two NFAs. Show that the product construction can be generalized to create an NFA $N = (Q_1 \times Q_2, \Sigma, \delta, s, A)$ such that $L(N_1) = L(N_1) \cap L(N_2)$. Your main task is to formally define δ, s, A in terms of the parameters of N_1 and N_2. Briefly justify why your construction works. A formal proof by induction is not needed.

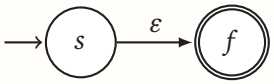
     Solution: Our approach is fundamentally similar to the product construction idea for DFAs. However, since, NFAs can contain ϵ-transitions, we must treat some edge cases involving these with special care.

     $$\begin{aligned} Q &:= Q_1 \times Q_2 \\ s &:= (s_1, s_2) \\ A &:= A_1 \times A_2 \\ \delta((q_1, q_2), a) &:= \delta_1(q_1, a) \times \delta_2(q_2, a) \\ \delta((q_1, q_2), \varepsilon) &:= \big(\delta_1(q_1, \varepsilon) \times \{q_2\}\big) \cup \big(\{q_1\} \times \delta_2(q_2, \varepsilon)\big) \end{aligned}$$

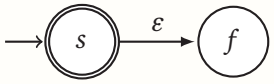
     ■

   - Let $N = (Q, \Sigma, \delta, s, A)$ be an NFA, and define the NFA $N_{\text{comp}} = (Q, \Sigma, \delta, s, Q \setminus A)$. In other words we simply complemented the accepting states of N to obtain N_{comp}. Describe a concrete example of a machine N to show that $L(N_{\text{comp}}) \neq \overline{L(N)}$. You need to explain for your machine N what $\overline{L(N)}$ and $L(N_{\text{comp}})$ are.

     Solution: Consider the following NFA N:

     Switching the accept and non accepting states in this NFA gives us N_{comp} as follows:

     $\varepsilon \in L(N)$, so $\varepsilon \notin \overline{L(N)}$. However, $\varepsilon \in L(N_{\text{comp}})$ (in fact $L(N) = L(N_{\text{comp}}) = \{\varepsilon\}$), so $\overline{L(N)} \neq L(N_{\text{comp}})$. ■

- Define an NFA or DFA that accepts $\overline{L(N)} - L(N_{\text{comp}})$, and explain how it works.

Solution: We first notice that for an NFA N, every string w may be classified into one of four types:

Type 1 w leads to no state in N. That is, every computation on w crashes.

Type 2 w leads to at least one accept state in N, but no non-accepting states.

Type 3 w leads to at least one non-accepting state in N, but no accepting states.

Type 4 w leads to at least one accepting state in N and also leads to at least one non-accepting state of N.

We now enumerate strings that are in $L(N)$ (i.e., accepted by N), $\overline{L(N)}$, and $L(N_{\text{comp}})$. Using our aforementioned classification:

- $L(N)$ contains strings of type 2 and 4, since N contains an accepting path for w in these cases.
- $\overline{L(N)}$ contains strings of type 1 and 3, since by definition, $\overline{L(N)} = \Sigma^* - L(N)$
- $L(N_{\text{comp}})$ contains strings of type 3 and 4, since if N contains a path for w to a non-accepting state, it will be turned into an accepting state in $\overline{N}$).

The set difference $\overline{L(N)} - L(N_{\text{comp}})$ contains strings of type 1 only. Hence, we want to accept all and only strings that lead nowhere in N. To do this, we build the powerset DFA M' that simulates N, but make the accepting state the one labeled with the empty set. More formally, we define the automaton $M' = (Q', \Sigma, \delta', s', A')$ by

- $Q' = P(Q)$.
- $\Sigma = \Sigma$.
- $\delta'(B, a) = \{\delta(q, a) \mid q \in B\}$.
- $s' = \{s\}$.
- $A' = \{\varnothing\}$.

One plausible but incorrect approach is to modify N to obtain M' as follows: make all states non-accepting, and add all edges missing in N to point to a new accepting state in M'. While this does let previously crashed computations in N be accepted in M', such strings might already have been accepted in N via another path, and so should be rejected in M'. ■

Solution: Let D and D_{comp} be the DFAs generated by applying the incremental subset construction to N and N_{comp}, respectively, so that $L(D) = L(N)$ and $L(D_{\text{comp}}) = L(N_{\text{comp}})$.

Applying the complement construction to D gives a DFA $\overline{D}$ so that $L(\overline{D}) = \overline{L(D)} = \overline{L(N)}$, and then applying the product construction for set difference to $\overline{D}$ and D_{comp} gives us a DFA M for the desired language. ■

- **Not to submit:** Define an NFA that accepts $L(N_{\text{comp}}) - \overline{L(N)}$, and explain how it works.

Solution: The set difference $L(N_{\text{comp}}) - \overline{L(N)}$ contains strings of type 4 (see above) only, so we can construct a DFA M'' that accepts exactly these strings by creating the powerset construction DFA M'', but a state (set of states belonging to N) C of the M'' is an accepting state if and only if it contains both an accepting and a non-accepting state of N. More formally, we define the automaton $M'' = (Q'', \Sigma, \delta'', s'', A'')$ by

- $Q'' = P(Q)$.
- $\Sigma = \Sigma$.
- $\delta''(B, a) = \{\delta(q, a) \mid q \in B\}$.
- $s'' = \{s\}$.
- $A'' = \{C : \exists p \in C \cap A \text{ and } \exists q \in C \cap (Q - A)\}$.

■

Solution: Let D and D_{comp} be the DFAs generated by applying the incremental subset construction to N and N_{comp}, respectively, so that $L(D) = L(N)$ and $L(D_{\text{comp}}) = L(N_{\text{comp}})$.

Applying the complement construction to D gives a DFA $\overline{D}$ so that $L(\overline{D}) = \overline{L(D)} = \overline{L(N)}$, and then applying the product construction for set difference to D_{comp} and $\overline{D}$ gives us a DFA M for the desired language. ■

Rubric: 10-points:

- 2 points for correctly stating the answer to part (a)
- 4 points for each of parts (b) and (c):
  - +2 for the main idea
  - +2 for correctly/accurately specifying the automaton.