CSE 103 Computational Models

Fall 2021 Prof. Fremont

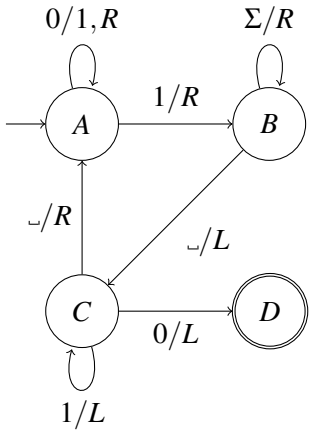
Practice 2

Final Practice Problems

This set of problems tries to cover most of the material we've seen in the second half of the class, so it is large. Feel free to jump around and find (sub)problems which you think will be most helpful. Each subproblem is a reasonable question to have on the final.

1. Working with a TM

Consider the following Turing machine M with $\Sigma = \{0,1\}$ (where we do not draw the reject state):



(a) List the computation history of M when run on input 0101.

Solution:

$A0101$

$1A101$

$11B01$

$110B1$

$1101B\sqcup$

$110C1$

$11C01$

$1D101$

(b) For each of the following input strings, state whether M accepts, rejects, or does not halt.

- ε
- 1
- 10

Solution:

- ε: This is rejected: in the start configuration, the symbol under the head is $\sqcup$, for which the graph above does not show a transition, so we move to the reject state.
- 1: M does not halt on this input: from the start state we read 1, move right, and go to B; then we read a blank, move left, and go to C; then read 1, move left, and stay in C; then read a blank, move right, and go to A. Now we are in the same state we started in, with the same tape contents, so the sequence above will repeat forever.
- 10: This is accepted: M reads the 1, goes to B, moves past the end of the input then left to the last symbol and state C; then it reads the 0 and accepts.

(c) What is the language of M?

Solution: From the examples above (and confirming with the graph), we can see that state A causes us to move past all 0s at the start of the input, converting them to 1s. When we see the first 1, we go to state B, which moves past the rest of the input, leaving it unchanged. Then we enter C on the last input symbol, and move left past all 1s; if we find a 0, we accept, whereas if we get back to the start of the input (and so read a blank) we return to state A. So if the input string has any 0 after the first 1, we will accept; otherwise, after leaving state A the tape will contain only 1s and M will run forever. So $L(M) = 0^*1\Sigma^*0\Sigma^*$.

2. Designing TMs

Design TMs deciding each of the following languages (not worrying about runtime). Give high-level verbal descriptions, like the solution to Problem 3 on HW 6.

(a) Binary strings with twice as many 0s as 1s.

Solution: We'll make successive passes through the input, crossing off two 0s every time we see a 1. Then if we eventually have no symbols left, there are exactly twice as many 0s as 1s.

(1) Scan the entire input to find a 1. If one is found, cross it off and continue to step (2). Otherwise, scan the input to find a 0: if there is one, reject, and otherwise accept.

(2) Scan the entire input to find a 0, then cross it off and find another 0; cross it off as well. If either 0 cannot be found, reject. Otherwise return to step (1).

(b) Binary strings of the form $x\overline{x}$, where $\overline{x}$ is x with all its bits flipped (e.g. $\overline{001} = 110$).

Solution: First, we'll find the middle of the string, where the transition from x to $\overline{x}$ should take place. Then we can use the usual back-and-forth traversal to compare the first and second halves of the string, except that we match 0s in the first half to 1s in the second half and vice versa.

(1) Mark the first symbol, then scan to the end of the input and mark the last symbol. Return to the first unmarked symbol and repeat, so that we mark two symbols on each pass. If we find there are no unmarked symbols left, then use a different mark to indicate the last symbol marked as the beginning of the second half. If there is only one unmarked symbol left, then the input has odd length, so we reject.

(2) Go through the symbols of the first half, crossing off each one in turn so that we don't process it again. For each symbol x, use states to remember whether it is a 0 or a 1, then proceed to the second half of the string and find the corresponding symbol y (i.e. the leftmost one which is not yet crossed off). If $x = y$, reject; otherwise cross off y and continue the iteration. Once every symbol of the first half is crossed off, accept.

(c) Strings over $\Sigma = \{0, 1, \$\}$ of the form $x\$y$ where $x, y \in \{0,1\}^*$ and x is a substring of y.

Solution: Our TM will have two stages: first, we'll check that the input has the form $x\$y$; second, we'll compare x against y starting from the first symbol of y, then the second symbol, etc., looking for a match at all possible positions.

(1) Scan from left to right looking for the \$ and then continuing on to the end of the input: if we get through the input without finding a \$, or find multiple \$s, reject. If x is empty (i.e. the \$ is the first symbol), we accept immediately since it is trivially a substring of y.

(2) Compare the symbols of x with those of y, marking the symbols we've already seen (e.g. replacing 0 with $\hat{0}$) and using states to remember the symbol of x we're currently comparing. If all symbols of x have counterparts in y, we accept. If we run off the end of y, then a match is not possible and we reject. Otherwise, x is not a prefix of y, but could appear as a substring later on in y.

(3) Cross off the first symbol of y (say using an X symbol) and unmark all other symbols. Repeat step (2), except skipping over all crossed-off symbols. This will compare x against y starting from the second position. Once that comparison is complete, we cross off the next symbol of y, and continue until we either accept or reject in step (2) (which must happen eventually, since once enough of y has been crossed off the remaining part will be shorter than x and we will reject).

3. Properties of reductions and decidable and RE languages

For each of the following statements, state whether it is true or false and give a 1-3 sentence justification.

(a) If $A \leq_m B$, then $\overline{A} \leq_m \overline{B}$.

Solution: This is true: if f is the mapping reduction from A to B, then $x \in A \iff f(x) \in B$, so $x \in \overline{A} \iff f(x) \in \overline{B}$. So f is also a mapping reduction from $\overline{A}$ to $\overline{B}$.

(b) Every language is either RE or co-RE.

Solution: This is false: there are languages which are neither RE nor co-RE, like EQ_{TM} (see Problem 5 below).

(c) A language L is decidable if and only if $L \leq_m \{010\}$.

Solution: This is true. If L is decidable, then the function

$$f(x) = \begin{cases} 010 & x \in L \\ 1 & \text{otherwise} \end{cases}$$

is computable, and gives a mapping reduction from L to $\{010\}$. Conversely, given such a mapping reduction f, we can decide whether $x \in L$ by computing $f(x)$ and checking whether the result is the string 010.

(d) If $L_0, L_1, L_2, \ldots$ is a sequence of RE languages, then $L = \cup_{i \geq 0} L_i$ is RE.

Solution: This is false: enumerate all the elements of $\overline{HALT}$ as $s_0, s_1, s_2, \ldots$. Then each language $L_i = \{s_i\}$ is RE (in fact decidable), since it contains only a single string. But $L = \cup_{i \geq 0} L_i = \overline{HALT}$ is not RE.

(e) If $A \leq_m B$, then $A \leq_m C$ for all $C \supseteq B$.

Solution: This is false: take any languages A and B over Σ where $A \leq_m B$ and there is some string $x \notin A$. Then $\Sigma^* \supseteq B$, but $A \not\leq_m \Sigma^*$, since any mapping reduction f would have to satisfy $f(x) \notin \Sigma^*$, which is impossible.

4. A closer look at the emptiness problem

Recall that $E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$.

(a) Prove that E_{TM} is co-RE.

Solution: We need to show that the complement,

$$\overline{E_{TM}} = \{x \mid x \text{ does not encode a TM, or it encodes a TM } M \text{ with } L(M) \neq \emptyset\},$$

is RE. We can build a recognizer R for $\overline{E_{TM}}$ as follows: given an input x, first check if it encodes a TM; if not, accept. Otherwise, x encodes some TM M with alphabet Σ, and we need to check whether its language is nonempty. Run M "in parallel" on all words in Σ^*, i.e. use the parallel simulation idea from class where we run simulations on N inputs for up to N steps, iteratively increasing N. If any of these simulations accepts, accept; otherwise continue searching.

If $x \in \overline{E_{TM}}$, then either x doesn't encode a TM, in which case we accept immediately, or x encodes a TM M whose language is nonempty, in which case we will eventually simulate M to completion on a word in $L(M)$ and accept. If instead $x \notin \overline{E_{TM}}$, then x encodes a TM M with $L(M) = \emptyset$, so none of our simulations will ever accept and so R will run forever. Therefore R accepts x if and only if $x \in \overline{E_{TM}}$, so R is a recognizer for $\overline{E_{TM}}$.

(b) Is E_{TM} Turing-recognizable? Why or why not?

Solution: It is not, since if it were recognizable it would be RE, and since we showed above that it is co-RE, it would then be decidable. But we proved in class that E_{TM} is undecidable.

(c) Is there a mapping reduction from A_{TM} to E_{TM}? If so, describe one; if not, argue why one cannot exist.

Solution: There is no such mapping reduction, even though there is a *Turing reduction* ($A_{TM} \leq_T E_{TM}$), as we showed in class (see Example 5.27 in the textbook for a discussion). If $A_{TM} \leq_m E_{TM}$, then by the argument in Question 3 (a) we would have $\overline{A_{TM}} \leq_m \overline{E_{TM}}$. Then since we showed above that E_{TM} is co-RE, $\overline{E_{TM}}$ is RE and applying the mapping reduction we would get that $\overline{A_{TM}}$ is also RE. This means that A_{TM} would be co-RE. However, we know that A_{TM} is RE, since it is the language of the universal TM, so then A_{TM} would be both RE and co-RE and therefore decidable. This is a contradiction, since we know A_{TM} is undecidable, so there cannot be a mapping reduction from A_{TM} to E_{TM}.

5. The equivalence problem for Turing machines

The equivalence problem for TMs is $EQ_{TM} = \{\langle M,N\rangle \mid M \text{ and } N \text{ are TMs and } L(M) = L(N)\}$.

(a) Prove that EQ_{TM} is not RE.

Solution: We'll prove that $\overline{HALT} \leq_m EQ_{TM}$: since $\overline{HALT}$ is not RE, this will show that neither is EQ_{TM}. If $\langle M,x\rangle \in \overline{HALT}$, then M runs forever on input x. We can design a TM N that accepts nothing in this case, otherwise accepting everything. Specifically, N does the following on input y:

(1) Run M on input x.

(2) Accept.

If $\langle M,x\rangle \in \overline{HALT}$, then N gets stuck on step (1) and so accepts no strings. If instead $\langle M,x\rangle \notin \overline{HALT}$, then step (1) finishes and N accepts all strings. Let $f(\langle M,x\rangle) = \langle N,A\rangle$ where A is a trivial TM rejecting all inputs. This function is a mapping reduction from $\overline{HALT}$ to EQ_{TM}, since $L(N) = L(A)$ if and only if $\langle M,x\rangle \in \overline{HALT}$.

(b) Prove that EQ_{TM} is not co-RE.

Solution: We'll follow the same approach as above to build a mapping reduction from $\overline{HALT}$ to $\overline{EQ_{TM}}$, showing that $\overline{EQ_{TM}}$ is not RE and so that EQ_{TM} is not co-RE. Given an instance $\langle M,x\rangle$ of $\overline{HALT}$, build a TM N in exactly the same way as above; as we showed there, N accepts all strings if and only if $\langle M,x\rangle \notin \overline{HALT}$. Let $f(\langle M,x\rangle) = \langle N,B\rangle$ where B is a trivial TM accepting all inputs. This function is a mapping reduction from $\overline{HALT}$ to $\overline{EQ_{TM}}$, since $L(N) \neq L(B)$ if and only if $\langle M,x\rangle \in \overline{HALT}$.

6. Decidable problems

Argue why each of the following problems is decidable.

(a) $COUNT_{DFA} = \{\langle D, n\rangle \mid D \text{ is a DFA and } |L(D)| = n\}$

Solution: To count the number of words in $L(D)$, we'll start by doing a graph traversal of D to find all the reachable states, deleting any which are not reachable. We can similarly prune away all states from which an accepting state is not reachable. If the start state is pruned away, then $L(D) = \emptyset$, so we'll accept only if $n = 0$. If the part of D remaining after pruning contains a cycle, then $L(D)$ is infinite (like in the pumping lemma: we can repeat the cycle as many times as desired to produce more strings in the language), so we'll reject. Otherwise, the graph is acyclic, so there are finitely-many paths from the start state to an accepting state. We can just enumerate all such paths, count them, and accept if there are exactly n.

(*Note:* In fact, you can even implement this algorithm in polynomial time, by using dynamic programming to count the paths without explicitly enumerating them. This is basically the same as the classical algorithm for counting paths in a DAG, if you've seen that before.)

(b) $L = \{\langle R\rangle \mid R \text{ is a regular expression matching only words of even length}\}$

Solution: We can decide L by first converting R to an equivalent DFA D (the conversion procedure we described in class can clearly be implemented by an algorithm), intersecting it with a DFA O that accepts all words of *odd* length, and applying the algorithm for E_{DFA}: if the intersection is empty, then R matches no words of odd length, so we accept; otherwise, we reject.

7. Undecidable problems

Prove that each of the following problems is undecidable.

(a) $L = \{\langle M \rangle \mid M \text{ is a TM that on any input either runs forever or halts in polynomial time}\}$

Solution: We will reduce $HALT$ to L. Given an instance $\langle M, x \rangle$ of $HALT$, let the TM N do the following on input y:

(1) Run M on input x.

(2) Run for $2^{|y|}$ steps.

If M halts on x, then N halts on every input y, but its runtime is at least $2^{|y|}$ and not polynomial, so $\langle N \rangle \notin L$. If instead M does not halt on x, then N runs forever on all inputs, so $\langle N \rangle \in L$. So we can decide whether $\langle M, x \rangle \in HALT$ by constructing N, asking the oracle whether $\langle N \rangle \in L$, and accepting if not. This shows that $HALT \leq_T L$, and since $HALT$ is undecidable, so is L.

(b) $L = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = SAT\}$

Solution: We will reduce $HALT$ to L. Given an instance $\langle M, x \rangle$ of $HALT$, let the TM N do the following on input y:

(1) Run M on input x.

(2) If $y \in SAT$, accept; otherwise reject.

If M halts on x, then N accepts y iff $y \in SAT$, so $L(N) = SAT$. If instead M does not halt on x, then N runs forever on every input, so $L(N) = \emptyset \neq SAT$. So the function $f(\langle M, x \rangle) = \langle N \rangle$ gives a mapping reduction from $HALT$ to L. Since $HALT$ is undecidable, so is L.

8. Polynomial-time decidable problems

For each of the following problems, argue why it is in P.

(a) $LONG_{DFA} = \{\langle D, 1^n\rangle \mid D \text{ is a DFA which only accepts words of length} \geq n\}$

Solution: If Σ is the alphabet of D, we can construct a DFA S such that $L(S) = \Sigma^{<n}$: we use a chain of n states to count how many symbols have been seen so far, and once we reach n we go into a rejecting sink state. Then to check if D accepts any words of length $< n$, we can use the product construction to intersect D and S and then apply the emptiness algorithm for DFAs. If $L(D) \cap L(S) = L(D) \cap \Sigma^{<n}$ is empty, then $\langle D, 1^n\rangle \in LONG_{DFA}$ and we accept; otherwise we reject. This procedure takes polynomial time, since S has size $O(n)$ and the product and emptiness algorithms take polynomial time. (In fact, this shows that $LONG_{DFA} \leq_P E_{DFA}$.)

(b) $FUNNY\text{-}CLIQUE = \{\langle G, k, 1^n\rangle \mid G = (V, E) \text{ is an undirected graph with a } k\text{-clique and } n = |V|^k\}$

(Notice the difference from *CLIQUE*, which we've proved is *NP*-complete.)

Solution: A k-clique is a set of k vertices. Since G has $|V|$ vertices, there are $\binom{|V|}{k} \leq |V|^k = n$ such sets; for each one, testing whether it is a k-clique can be done in polynomial time (we just look to see if G contains all the required edges). So checking all possible k-cliques can be done in time polynomial in $|G|$ and n, and thus polynomial in $|\langle G, k, 1^n\rangle|$.

(The point here is that the longer your input, the longer your algorithm can run while still being considered polynomial-time; here, the extra padding at the end of the input is exponentially long, allowing us to solve the inner instance of *CLIQUE* using an algorithm that takes time exponential in k while still being polynomial in the length of the entire input.)

9. NP problems

Prove that the following problems are in NP.

(a) $COMPOSITE = \{\langle n \rangle \mid n \text{ is a composite (non-prime) integer}\}$

Solution: If n is composite, then $n = ab$ for some integers $a, b < n$, and since we can do multiplication in polynomial time, we'll use the pair $\langle a, b \rangle$ as our certificate. Given the input $\langle n, y \rangle$, our verifier will check that y encodes a pair (a, b) of integers, that $a, b < n$, and that $ab = n$, accepting if so. Then if n is composite, there is a value of the certificate causing the verifier to accept; otherwise, $n \neq ab$ for any $a, b < n$ and so the verifier will always reject. Since the verifier runs in polynomial time, this shows that *COMPOSITE* is in NP.

(*Note:* This problem is actually also in P, but that was only proved in 2002!)

(b) $MAX\text{-}CUT = \{\langle G, k \rangle \mid G \text{ is an undirected graph with a cut of size} \geq k\}$ (where a *cut* is a partition of the vertices of G into two groups; its *size* is the number of edges between the groups.)

Solution: The certificate is the cut. Given the input $\langle G, k, y \rangle$, with $G = (V, E)$ an undirected graph, our verifier will first check that y encodes a cut, i.e. a pair (A, B) where $A, B \subseteq V$ and $A \cap B = \emptyset$. If so, and the number of edges $(u, v) \in E$ with $u \in A$ and $v \in B$ is at least k (which we can check in polynomial time by iterating over all the edges of G), the verifier accepts; otherwise it rejects. If G has a cut (A, B) of size $\geq k$, then the verifier will accept with certificate $y = \langle A, B \rangle$; otherwise, there is no cut of size $\geq k$ and so the verifier will reject for any value of y. Since the verifier runs in polynomial time, this shows that *MAX-CUT* is in NP.

(*Note:* This problem is actually also NP-complete, although the reduction to prove NP-hardness is a bit challenging. See Problem 7.27 in the textbook if you'd like to try it.)

10. NP-complete (or not) problems

For each of the following problems, state whether it is NP-complete or not under the assumption that $\mathsf{P} \neq \mathsf{NP}$ (since otherwise all nontrivial NP problems are NP-complete), and give a 1-3 sentence justification.

(a) $ALL_{DFA} = \{\langle D \rangle \mid D \text{ is a DFA with alphabet } \Sigma \text{ and } L(D) = \Sigma^*\}$

Solution: This is not NP-complete (if $\mathsf{P} \neq \mathsf{NP}$), since it is in P: to check whether a DFA accepts all strings we can complement it and apply the polynomial-time algorithm for E_{DFA}.

(b) *AT-MOST-3SAT* $= \{\langle \varphi \rangle \mid \varphi \text{ is a Boolean formula in CNF with } \leq 3 \text{ literals in each clause}\}$

Solution: This is NP-complete: we can reduce *3SAT* to it trivially, since every instance of *3SAT* is also an instance of *AT-MOST-3SAT*, so it is NP-hard. It is also in NP, being a special case of SAT.

(c) *5-CLIQUE* $= \{\langle G \rangle \mid G \text{ is an undirected graph with a 5-clique}\}$

Solution: This is not NP-complete (if $\mathsf{P} \neq \mathsf{NP}$), since it is in P: we can go through all sets of 5 vertices and check whether they form a clique. If G has v vertices, there are $\binom{v}{5} = O(v^5)$ such sets, so this procedure will take polynomial-time (noting that v is at most linear in the size of the input).

(d) $A_{TM} = \{\langle M, x \rangle \mid M \text{ is a TM that accepts the string } x\}$

Solution: This is not NP-complete, since it is undecidable and so not in NP.

(e) *BOUNDED-*$A_{TM} = \{\langle M, x, 1^n \rangle \mid M \text{ is a TM that accepts the string } x \text{ in at most } n \text{ steps}\}$

Solution: This is not NP-complete (if $\mathsf{P} \neq \mathsf{NP}$), since it is in P: we can simulate M on input x for n steps and see if it accepts within that time. The simulation takes time polynomial in n and the sizes of M and x, which is polynomial in the length of the input $\langle M, x, 1^n \rangle$.

(f) *BOUNDED-*$A_{NTM} = \{\langle M, x, 1^n \rangle \mid M \text{ is an NTM that accepts } x \text{ in at most } n \text{ steps along some branch}\}$ (where an NTM is a *nondeterministic* Turing machine)

(*Hint:* Use the fact that every NP problem has an NTM deciding it in polynomial time.)

Solution: This is NP-complete. It is in NP, since there is a polynomial-time verifier: the certificate is the sequence of $\leq n$ nondeterministic choices specifying the branch of M to simulate (alternatively, we can give an NTM deciding this language in polynomial time: it is a nondeterministic version of the universal TM which simulates M for n steps). It is also NP-hard: for any $A \in \mathsf{NP}$, there is an NTM M deciding A in time at most $p(n)$ for some polynomial p, and the function $f(x) = \langle M, x, 1^{p(|x|)} \rangle$ gives a polynomial-time reduction from A to *BOUNDED-*A_{NTM}.

11. NP-hardness proofs

Prove each of the following problems is NP-hard by reducing a problem we already know is NP-hard to it.

(a) $L = \{\langle G, H\rangle \mid G$ and H are undirected graphs, and some subgraph of G is isomorphic to $H\}$
(i.e. G contains a copy of H, up to relabeling vertices; this is called the *subgraph isomorphism problem*)

Solution: We'll reduce *CLIQUE* to L. Given an instance $\langle G, k\rangle$ of *CLIQUE*, let H be the complete graph with k vertices (i.e. the graph with k vertices all connected to each other). A k-clique in G is isomorphic to H, so $\langle G, k\rangle \in$ *CLIQUE* $\iff \langle G, H\rangle \in L$. We can build H in polynomial time, since k is at most the number of vertices of G, so the function $f(\langle G, k\rangle) = \langle G, H\rangle$ is a polynomial-time reduction from *CLIQUE* to L.

(b) *COUNT-SAT* $= \{\langle \varphi, k\rangle \mid \varphi$ is a Boolean formula with at most k satisfying assignments$\}$
(Side note: this problem is thought to *not* be in NP (or co-NP), but in a higher complexity class. It's usually viewed as a counting problem called *#SAT* rather than a decision problem.)

Solution: We'll reduce *SAT* to *COUNT-SAT*. Given a *SAT* instance $\langle \varphi \rangle$, consider the negated formula $\overline{\varphi}$ (that is, the formula ϕ with an extra NOT operator applied). Notice that a satisfying assignment of φ does *not* satisfy $\overline{\varphi}$ and vice versa. So if φ has a satisfying assignment, then $\overline{\varphi}$ has at least one non-satisfying assignment, and so at most $2^n - 1$ satisfying assignments if φ has n variables (since there are 2^n assignments total). On the other hand, if φ is unsatisfiable, then every assignment satisfies $\overline{\varphi}$, so $\overline{\varphi}$ has 2^n satisfying assignments. Therefore φ is satisfiable if and only if $\langle \overline{\varphi}, 2^n - 1\rangle \in$ *COUNT-SAT*. Since we can compute $\overline{\varphi}$ and $2^n - 1$ in polynomial time, the function $f(\langle \varphi \rangle) = \langle \overline{\varphi}, 2^n - 1\rangle$ gives a polynomial-time reduction from *SAT* to *COUNT-SAT*.

12. Pumping lemma for context-free languages

Prove that the following languages are not context-free:

(a) $L = \{a^i b^j a^i b^j \mid i, j \geq 1\}$

Solution: Given $p \geq 1$, we'll pick the word $w = a^p b^p a^p b^p$, which satisfies $w \in L$ and $|w| = 4p \geq p$. Now for any decomposition $w = uvxyz$ with $|vy| > 0$ and $|vxy| \leq p$, since the blocks of each letter have length p, the string vxy can only overlap at most two consecutive blocks. In particular, it is not possible for vxy to overlap both blocks of as or both blocks of bs. Furthermore, since $|vy| > 0$, the string vy contains either an a or a b (or both). So setting $k = 2$, the string $uv^k xy^k z$ has more as or bs than w. If the new symbols cause the string to no longer have the required 4-block form, then it is not in L. Otherwise, the length of at least one of the blocks has increased, but the length of the corresponding block with the same letter has not changed (since vxy can't overlap both blocks), so again the string is not in L. Therefore in all cases $uv^2xy^2z \notin L$, and so by the contrapositive of the pumping lemma L is not context-free.

(b) $L = \{0^i 1^i 2^j \mid i, j \geq 0 \text{ and } i \leq j\}$

Solution: Given $p \geq 1$, we'll pick the word $w = 0^p 1^p 2^p$, which satisfies $w \in L$ and $|w| = 3p \geq p$. Now for any decomposition $w = uvxyz$ with $|vy| > 0$ and $|vxy| \leq p$, since the blocks of each digit have length p, the string vxy cannot contain all three digits. So if vy contains a 0, then it cannot contain a 2, and setting $k = 2$ yields a string $uv^k xy^k z$ with strictly more 0s but the same number of 2s as w. Therefore $uv^k xy^k z$ has strictly more 0s than 2s (since there were equal numbers of these in w) and is not in L. If instead vy does *not* contain a 0, then it must contain either a 1 or a 2 (or both) since $|vy| > 0$. Then setting $k = 0$, the string $uv^k xy^k z = uxz$ has strictly fewer 1s or 2s than w, but the same number of 0s: so as above it has strictly fewer 1s or 2s than 0s, and again is not in L. Therefore in all cases there is a $k \geq 0$ such that $uv^k xy^k z \notin L$, and so by the contrapositive of the pumping lemma L is not context-free.

CSE 103 Computational Models
Fall 2021 Prof. Fremont HW 6

(3 questions, 280 points total)

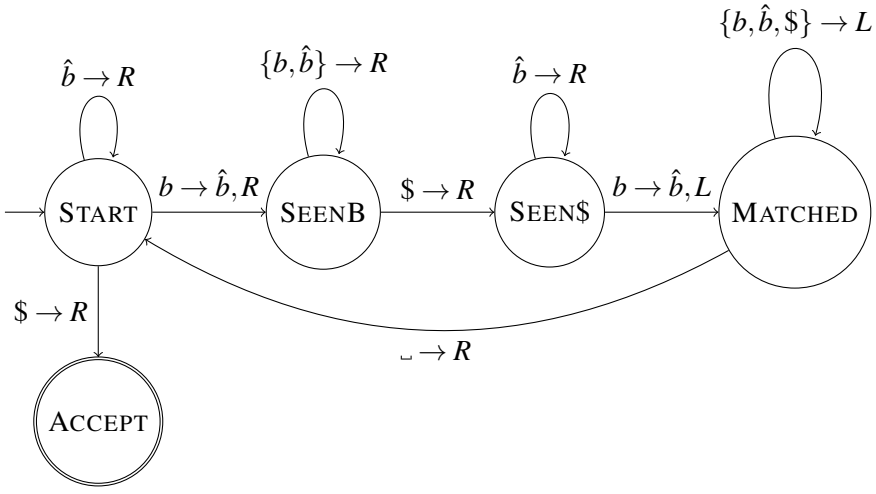
1. (100 pts.) Working with a TM

Consider the Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where:

- $Q = \{\text{START}, \text{SEENB}, \text{SEEN\$}, \text{MATCHED}, \text{ACCEPT}, \text{REJECT}\}$
- $\Sigma = \{b, \$\}$
- $\Gamma = \{b, \hat{b}, \$, \sqcup\}$
- $\delta(q,s) = \begin{cases} (\text{ACCEPT}, s, R) & q = \text{START and } s = \$ \\ (\text{SEENB}, \hat{b}, R) & q = \text{START and } s = b \\ (\text{SEEN\$}, s, R) & q = \text{SEENB and } s = \$ \\ (q, s, R) & q = \text{SEENB and } s = b \\ (\text{MATCHED}, \hat{b}, L) & q = \text{SEEN\$ and } s = b \\ (\text{START}, s, R) & q = \text{MATCHED and } s = \sqcup \\ (q, s, L) & q = \text{MATCHED and } s \in \{b, \hat{b}, \$\} \\ (q, s, R) & s = \hat{b} \\ (\text{REJECT}, s, R) & \text{otherwise} \end{cases}$
- $q_0 = \text{START}$
- $q_{\text{accept}} = \text{ACCEPT}$
- $q_{\text{reject}} = \text{REJECT}$

(a) (*45 pts.*) Draw M as a graph, omitting the reject state and all transitions leading to it (following the convention used in class, we will say that all missing transitions lead to the reject state).

Solution:



(Note that labels like $\{b, \hat{b}\} \to R$ are just shorthand for the two transitions $b \to R$ and $\hat{b} \to R$, which in turn are shorthand for $b \to b, R$ and $\hat{b} \to \hat{b}, R$; any of these notations is fine.)

(b) (*15 pts.*) List the sequence of configurations of M (the computation history) when run on input $b\$bb$. Please put each configuration on a separate line.

Solution:

$$\begin{aligned}
&\textsc{Start}\ b\ \$\ b\ b\\
\hat{b}\ &\textsc{SeenB}\ \$\ b\ b\\
\hat{b}\ \$\ &\textsc{Seen\$}\ b\ b\\
\hat{b}\ &\textsc{Matched}\ \$\ \hat{b}\ b\\
&\textsc{Matched}\ \hat{b}\ \$\ \hat{b}\ b\\
&\textsc{Matched}\ \textvisiblespace\ \hat{b}\ \$\ \hat{b}\ b\\
&\textsc{Start}\ \hat{b}\ \$\ \hat{b}\ b\\
\hat{b}\ &\textsc{Start}\ \$\ \hat{b}\ b\\
\hat{b}\ \$\ &\textsc{Accept}\ \hat{b}\ b
\end{aligned}$$

(c) (*20 pts.*) For each of the following input strings, state whether M accepts, rejects, or does not halt.

- ε
  Solution: This is rejected, since the symbol under the head is a blank, and $\delta(\textsc{Start}, \textvisiblespace) = (\textsc{Reject}, \textvisiblespace, R)$, so we transition to the reject state and halt.
- $bb\$b$
  Solution: This is rejected: following a similar pattern to the computation history above, M will mark the first b on either side of the \$, reaching the configuration $\textsc{Start}\ \hat{b}\ b\ \$\ \hat{b}$. It will then proceed to mark the remaining b and move past the \$ and the $\hat{b}$, reaching the configuration $\hat{b}\ \hat{b}\ \$\ \hat{b}\ \textsc{Seen\$}$. But now the symbol under the head is a blank, and in state SEEN\$ this causes a transition to the reject state.
- $bb\$bbb$
  Solution: This is accepted, along the same lines as $b\$bb$ above. Each pass through the four states in the top row of the graph shown in part (a) marks the first b on each side of the \$, so that after two passes we will be in the configuration $\textsc{Start}\ \hat{b}\ \hat{b}\ \$\ \hat{b}\ \hat{b}\ b$. Then M reads past the $\hat{b}$s to get to the configuration $\hat{b}\ \hat{b}\ \textsc{Start}\ \$\ \hat{b}\ \hat{b}\ b$, whereupon M reads the \$ and transitions to the accept state.
- $\$b$
  Solution: This is accepted: the symbol under the head is a \$, so we transition immediately to the accept state.

(d) (*20 pts.*) What is the language of M?

Solution: As the examples above suggest, the behavior of the four main states is to find a b to the left of the \$, mark it, then proceed past the \$ and find another b to mark (rejecting if there is no \$ at all, or we find a second \$ before finding a b). If there is no other b to mark, we will reach the end of the tape in state SEEN\$ and reject. Otherwise, we mark the b, and the MATCHED state brings us back to the start of the input and we repeat. Whenever one of these passes succeeds (i.e. does not reject partway through), we mark one b on each side of the \$. Once there are no unmarked bs to the left of the \$, the START state will move past all $\hat{b}$s and reach the \$, then moving to the accept state. So M accepts if and only if the input string starts with a prefix of the form $b^n\$b^m$ where $m \geq n$. Formally, we have $L(M) = \{b^n\$b^m x \mid m \geq n \geq 0 \text{ and } x \in \Sigma^*\}$.

2. (80 pts.) Fleshing out a TM description

Consider the following language over $\Sigma = \{0,1,\#,\$\}$, which represents a simple form of array lookup, namely checking if the kth element of the array $(e_0, e_1, \ldots, e_n)$ is equal to a given binary string v:

$$L = \{1^k \# v \$ e_0 \$ e_1 \$ \ldots \$ e_n \$ \mid 0 \le k \le n,\ v, e_0, \ldots, e_n \in \{0,1\}^*, \text{ and } e_k = v\}.$$

(So for example 11#011\$0\$100\$011\$11\$ $\in L$ but 11#011\$0\$100\$010\$11\$ $\notin L$.)

Consider the following medium-level description (what the textbook calls an "implementation description") of a TM deciding L, which uses the tape alphabet $\Gamma = \{0,1,\#,\$,X,\sqcup\}$ where X is a new symbol representing a "crossed-off" part of the tape:

1. Go through each of the 1s at the start of the input; for each one, cross it off and scan right until you find a \$, and cross that off as well, rejecting if there are no \$s left. (After this step, the leftmost remaining \$ will be the one just before e_k.)
2. Go through each of the symbols of v; for each one, remember whether it is a 0 or a 1, then scan right until you find a \$. Scan right until you find a 0 or 1, and reject if it doesn't agree with the symbol of v we saw earlier.
3. Go back to the #, then move right to the first \$. Continue moving right, and if you see another \$ without encountering any 0s or 1s along the way, then accept.

Let's flesh out this description into a more detailed one listing all the states and what should be done at each one. Use English, like the low-level description given in the 11/8 lecture (on page 2 of the notes): do not write out the TM as a graph.

(a) (*20* pts.) Describe how to implement step (1) above. You should only need 3 states, but don't worry about having exactly this number.

Solution:

We'll use three states, A, B, and C, which will handle finding the next 1, finding the corresponding \$, and returning to the start of the input respectively.

A: Move right past all Xs (so that we ignore all 1s we've previously handled and crossed out). If we find a #, then we've finished going through the start of the input and we'll move on to part (b) of this question. If we find a 1, then replace it with an X and go to state B, which will handle finding the corresponding \$. Otherwise, we find a 0, \$, or blank, none of which are allowed in this part of the input, so we'll reject.

B: Move right past all symbols except \$ and blanks. If we reach a blank, then we read through the whole input without finding a \$, so we reject. Otherwise we replace the \$ with an X and go to state C, which will handle returning to the first part of the input so we can continue going through the 1s.

C: Move left past all symbols except blanks. Once we reach a blank, we have moved all the way past the input, so we move right (leaving the head at the beginning of the input) and go to state A to proceed.

(b) (*40* pts.) Describe how to implement step (2) above, assuming the head of the TM is already at the first symbol of v. It's possible to do this with 6 states.

(*Hint:* If you're having trouble figuring this out, Example 3.9 in the textbook may be helpful.)

Solution:

We'll start with a state A which handles finding the next symbol of v, similarly to state A above. There is one important difference, however: we can't use the symbol X to cross off symbols of v we've

already read, since we may have used X to cross out the \$ after v and so we'd end up reading into e_0 by mistake. So we'll use the blank symbol ␣ instead.

Next, we'll have a state B to find the \$ and a state C to find the next 0 or 1; however, since we need to remember whether the symbol we saw in state A was a 0 or a 1, we'll actually have two copies B_0, C_0 and B_1, C_1 of these states. Finally, we'll have a state D to handle returning to the start of v so we can continue iterating through its symbols.

A: Read past all blanks to skip the bits of v we've already handled. If we find a \$ or an X, then we've finished going through v and we'll move on to part (c) of this question. If we find a #, the string is not of the right form (there should only be a # before v), so we reject. Otherwise we find a 0 or a 1; we replace it with a blank and go to state B_0 or B_1 accordingly.

B_0: Move right until we find a \$. If we encounter a blank, then we've made it past the entire input without finding a \$, so we reject; we also reject if we find a #, since the string then has multiple #s. Otherwise we find a \$, so we move right and go to state C_0.

C_0: Move right past all Xs (to ignore bits of e_k that we've already handled). If we find a 0, then this matches the 0 we saw earlier in v, so we replace it with X, move left, and go to state D. If we find a 1, then $v \neq e_k$, so we reject. If we find a \$ or a blank, then e_k is shorter than v, so we also reject. If we find a #, then the string has multiple #s, so we reject.

B_1: The same as B_0, except we go to state C_1 if we find a \$.

C_1: The same as C_0, except with 0 and 1 exchanged: if we find a 1, it matches the 1 we saw earlier, so we blank it out, move left, and go to state D; if we find a 0, we reject.

D: Move left until we find a #, then move right and go to state A.

(c) (*20* pts.) Describe how to implement step (3) above. It's possible to do this with 4 states.

Solution: This is similar to part (a). We'll use a state A to go back to the #, B to search for the first \$, and C to scan for any additional \$s.

A: Move left until we find the # (which the earlier parts of the problem have confirmed to be unique), then move right and go to state B.

B: Move right until we find a \$ (which we're guaranteed to do if we've gotten this far); then move right and go to state C.

C: Move right past all Xs. If we find a 0 or a 1, then e_k is longer than v, so we reject. If we find a #, the string has multiple #s, so we reject. If we find a blank, then the string is not terminated by a \$ as required, so we reject. Finally, if we find a \$, then we accept.

3. (100 pts.) Designing a TM

Design a TM that can compare two integers represented in binary; more precisely, that recognizes the language L of strings $x\$y$ where $x, y \in \{0,1\}^*$, $|x| = |y|$ (for simplicity), and $x \leq y$ when interpreted as integers. For example, 010\$100 and 001\$010 are in L but 1\$0 and 0\$00 are not. Give a medium-level description of your machine, like the 3-step description in the statement of Problem 2 above.

Solution: First we will check that the input has the right form, namely $x\$y$ with $x, y \in \{0,1\}^*$ and $|x| = |y|$. We can do this using a similar procedure to that of Problem 1, making several passes through the input, marking one unmarked symbol of both x and y at a time. To determine whether $x \leq y$ as integers, we'll again iterate through their bits: if the MSB of x is less/greater than the the MSB of y, then x is less/greater than y, so we can accept/reject immediately. If instead the MSBs are the same, then we'll continue on to the second bit, since we can't yet tell which of x or y is larger (or if they're the same). If we get through all the bits of x and y without ever finding one bit to be different from its counterpart, then $x = y$ and we'll accept. More precisely:

1. For each symbol before the \$, mark it and move past the \$, marking the next unmarked symbol. We reject if we do not encounter \$ at all, if there are no unmarked symbols left beyond the \$ (which would mean $|y| < |x|$), or if we encounter a second \$.
2. Once all the symbols before the \$ are marked, scan the symbols after the \$ and reject if any are unmarked (since this would mean $|x| < |y|$) or are a \$ (since this would mean the input contains multiple \$s). Otherwise, steps (1) and (2) ensure that the string has the form $x\$y$ with $x, y \in \{0,1\}^*$ and $|x| = |y|$.
3. Go through each bit before the \$, i.e., each bit of x. For each one, remember whether it is a 0 or a 1, then move to the corresponding bit after the \$, i.e., the corresponding bit of y (we can mark or cross off symbols as we process them to identify the next bit to go to). If the bit of x was 0 and the bit of y is 1, accept. If the bit of x was 1 and the bit of y is 0, reject. Otherwise the bits are equal, so we can't yet tell which of x or y is greater, so we continue.
4. If we get through all bits of x without rejecting, then $x = y$, so we accept.

CSE 103 Computational Models
Fall 2021 Prof. Fremont HW 7

(4 questions, 270 points total)

From this point in the course on, when defining a decision procedure, enumerator, etc., describing your algorithm is enough. You don't need to show how to actually implement it on a TM: we will assume the Church-Turing thesis. Just make sure to be precise: you don't need to give formal pseudocode (although you can if you think it would be clearer), but each step of your algorithm should be explained in enough detail that it's clear how the algorithm could be implemented on a computer.

1. (40 pts.) Short answer

For each of the following statements, state whether it is true or false and give a 1-2 sentence justification.

(a) If a language L and its complement $\overline{L}$ are both co-Turing-recognizable, then L is decidable.

Solution: This is true. Since L is co-Turing-recognizable, it is co-RE, as we've proved that a language is RE iff it is Turing-recognizable. Since $\overline{L}$ is also co-Turing-recognizable, its complement $\overline{\overline{L}} = L$ is Turing-recognizable and so RE. Therefore L is both RE and co-RE, and so it is decidable.

(b) If a language L is RE, then every subset of L is also RE.

Solution: This is false. For any alphabet Σ, the language Σ^* is RE: in fact it is decidable (by a TM which always accepts). So if the claim were true, then *every* language would be RE, whereas we've proved in class that some languages are not RE (e.g. $\overline{HALT}$).

2. (80 pts.) Decidable languages

Argue why each of the following languages is decidable.

(a) $L = \{\langle N_1, \ldots, N_k \rangle \mid \text{each } N_i \text{ is an NFA with alphabet } \Sigma \text{ and } \cup_i L(N_i) = \Sigma^*\}$

Solution: First, we check if the input has the form $\langle N_1, \ldots, N_k \rangle$ with each N_i being an NFA with the same alphabet; if not, we reject. Then, using our NFA constructions for union and complement (which can both be implemented as algorithms on a TM), we can build an NFA N such that $L(N) = \overline{\cup_i L(N_i)}$. Now $\cup_i L(N_i) = \Sigma^*$ if and only if $L(N) = \emptyset$, and we can decide the latter using the algorithm for E_{NFA} (the emptiness problem for NFAs).

Note: This argument shows that $L \leq_m E_{NFA}$: the reduction is given by the function $f(\langle N_1, \ldots, N_k \rangle) = N$, which is both computable and satisfies $\langle N_1, \ldots, N_k \rangle \in L \iff f(\langle N_1, \ldots, N_k \rangle) \in E_{NFA}$. (Strictly speaking, f also needs to be defined on inputs which aren't of the correct form $\langle N_1, \ldots, N_k \rangle$: for those, we can have f output some fixed NFA that has nonempty language.)

Alternate Solution: You could also check whether $\cup_i L(N_i) = \Sigma^*$ by constructing an NFA for the union, converting it to an equivalent DFA (which you had to do anyway above in order to take the complement), and then performing DFA minimization (which we did not cover in class, and which I do *not* expect you to know an algorithm for). Since the smallest DFA for the language Σ^* consists of a single accepting state with a self-loop, if this DFA is the result of minimization then $\cup_i L(N_i) = \Sigma^*$, and otherwise not.

(b) $L = \{\langle M, n \rangle \mid M \text{ is a TM that always halts within } n \text{ steps, on any input}\}$

(*Hint:* There are infinitely-many possible inputs, so you can't check them all. But do you need to check the inputs of length greater than n?)

Solution: Since the head of a TM reads one symbol and moves one cell left or right in each step of execution, in n steps it can only read the first n symbols of its input. So if a TM halts within n steps on an input x of length n, it also halts within n steps on any input string that begins with x (with arbitrary symbols afterward). Therefore, to check whether M halts within n steps on all inputs, it suffices to check whether M halts within n steps on all inputs of length at most n. We can decide this in finite time by going through the finitely-many strings of length at most n and simulating M on each one for up to n steps to see whether it halts within that time.

3. (100 pts.) Recursively-enumerable languages

Argue why each of the following languages is RE.

(Note: remember our theorem that being RE and being Turing-recognizable are equivalent; for each language below, it may be easier to prove one than the other.)

(a) Let L be the language of pairs of CFGs whose languages have at least three words in common, i.e. $\{\langle G_1, G_2\rangle \mid G_1 \text{ and } G_2 \text{ are CFGs and } |L(G_1) \cap L(G_2)| \geq 3\}$. Show that L is RE.

Solution: We can build a recognizer R for L as follows. Given an input string, reject if it is not of the form $\langle G_1, G_2\rangle$. Otherwise, enumerate all words in Σ^*: for each one, run the CYK algorithm (equivalently, the decision procedure for A_{CFG}) twice to check whether it is in both $L(G_1)$ and $L(G_2)$; if it is, increment a counter. If the counter reaches 3, accept; otherwise, continue enumerating words.

Any word in $L(G_1) \cap L(G_2)$ will eventually be enumerated, so if $|L(G_1) \cap L(G_2)| \geq 3$ then our counter will reach 3 and R will accept. If instead $|L(G_1) \cap L(G_2)| < 3$, then the counter will never reach 3, so R will run forever. Therefore R recognizes L, so L is Turing-recognizable and therefore RE.

(*Note:* It would not work to try to build a CFG whose language is the intersection of those of G_1 and G_2, since as we showed on HW 5, the context-free languages are not closed under intersection. Therefore it is possible that $L(G_1) \cap L(G_2)$ is *not* context-free.)

(b) *(60 pts.)* A *rewriting system* is a generalization of a CFG with the same components, namely a set of variables V, terminal symbols Σ, rules (or productions) P, and start symbol S, but now allowing the rules to have arbitrary strings on their left-hand sides instead of a single variable. For example, we could have a rule $xB \to xC$ which would allow rewriting the string $zxBz$ into $zxCz$ (notice that this rule is not "context-free", since it allows us to rewrite B only when it is preceded by an x), and even more complicated rules like $ABC \to DE$ are allowed.

As in the case of CFGs, the language $L(R)$ of a rewriting system R is the set of strings over Σ^* that can be derived from the start symbol S by applying the rules. Show that $L(R)$ is RE for any rewriting system R.

Solution: We can build an enumerator for $L(R)$ as follows: we'll keep track of a set D of all strings (of terminal and nonterminal symbols) we've derived so far. Initially we set $D = \{S\}$. For every string $w \in D$, we go through the rules P, applying them to w in every possible way: i.e., if the rule has the form $\alpha \to \beta$, we find all occurrences of α in w; we derive a new string for each one by replacing the occurrence of α with β. If the new string consists only of terminal symbols, we print it out. Finally, we add all newly-derived strings to D, and repeat this process.

Since we apply all the rules in all possible ways, any string that can be derived from S will eventually be added to D, and printed out if it consists only of terminals. Conversely, any string which cannot be derived from S will never be printed out. So this procedure will print out exactly the strings in $L(R)$, and is therefore an enumerator for $L(R)$. This shows that $L(R)$ is RE.

(*Note:* While $L(R)$ has to be RE, it *doesn't* have to be decidable, as was the case for context-free grammars. For any TM, you can come up with a rewriting system whose language is the same as the TM: use the rewrite rules to simulate how configurations of the TM change in each step of execution. This shows that if you take away the "context-free" restriction on grammars and allow arbitrary rewrite rules, they become as powerful as Turing machines. In particular, since *HALT* is Turing-recognizable, you could build a rewriting system for it, and this would be an example of a rewriting system with an undecidable language.)

4. (50 pts.) Using reductions to prove undecidability

Prove the following problem is undecidable by reducing one of the undecidable languages we saw in class to it: given a TM M, does M write more 1s than 0s on its tape when running on an empty input?

Solution: We will give a reduction from the halting problem, $HALT$. Given an input $\langle M,x\rangle$ for the halting problem, we construct a TM N that does the following:

(1) Simulate M on input x, but using new symbols not used by M in place of 0 and 1 so that the simulation does not require us to write any 0s or 1s.

(2) If the simulation finishes, write a single 1 on the tape and then accept.

Now if M halts on input x, then N will write a single 1 on its tape and no 0s. If instead M does not halt on input x, then N will not write any 0s or 1s on its tape. So we can decide whether $\langle M,x\rangle \in HALT$ by constructing N and checking whether it writes more 1s than 0s on its tape when run on empty input. Since $HALT$ is undecidable, so is the latter.

Note: We described the reduction above as a Turing reduction: a way to decide $HALT$ given an oracle for the language described in the problem statement (let's call it L). But it is also a mapping reduction: defining $f(\langle M,x\rangle) = \langle N\rangle$, the function f is computable, and we have $\langle M,x\rangle \in HALT \iff f(\langle M,x\rangle) \in L$ (here, as usual we've swept under the rug the detail that f needs to be defined for *all* inputs, not just those of the form $\langle M,x\rangle$: for malformed inputs we can just have f output some constant string that is not in L). Therefore $HALT \leq_m L$.

CSE 103 Computational Models
Fall 2021 Prof. Fremont HW 8

(4 questions, 260 points total)

1. (80 pts.) Mapping reductions and proving a language is not RE
Let $L = \{\langle M \rangle \mid M \text{ is a TM and } \varepsilon \notin L(M)\}$.

(a) Prove that $E_{TM} \leq_m L$.

Solution: Given a TM M, we need to build a TM N such that $\langle M \rangle \in E_{TM} \iff \langle N \rangle \in L$, i.e., M has empty language if and only if N does *not* accept the empty string. Equivalently, if M accepts *something* then we need to ensure that N accepts ε, and if M accepts nothing, then N should not accept ε. We can arrange this by defining N as follows:

1. Simulate M "in parallel" on all possible inputs. (As usual, we do this by running a finite number of simulations for a finite time each, iteratively increasing the time bound and the number of inputs.)
2. If any of the simulations end with M accepting, accept.

Now if $L(M) \neq \emptyset$, then one of the parallel simulations will complete with M accepting, so N will accept regardless of what its input was, and in particular it will accept ε. If instead $L(M) = \emptyset$, then none of the simulations will accept, so step (1) will run forever as it tries more and more inputs, and therefore N will not accept ε. So we have $\langle M \rangle \in E_{TM} \iff \langle N \rangle \in L$, as desired.

Furthermore, given $\langle M \rangle$, we can compute $\langle N \rangle$ (the "source code" for N just embeds the description of M, plus some extra code to orchestrate the parallel simulations), so setting $f(\langle M \rangle) = \langle N \rangle$ defines a computable function. Therefore f is a mapping reduction from E_{TM} to L.

(*Note:* Again, I've ignored the case where the input to f is not of the form $\langle M \rangle$ with M a TM. We can handle this in the usual way: f detects when the input x is not of this form, and outputs some constant string that is not in L. Then we will have $f(x) \notin L$, which is what we want since $x \notin E_{TM}$.)

(b) Prove that E_{TM} is not RE, which together with (a) implies that L is not RE either.

Solution: Observe that E_{TM} is co-RE: if a TM has *nonempty* language, we can detect that by simulating it in parallel on all possible inputs as above, accepting if any parallel simulation does. This gives a recognizer for $\overline{E_{TM}}$, so it is RE and therefore E_{TM} is co-RE. Now if E_{TM} were also RE, then it would be both RE and co-RE and so decidable. But we know E_{TM} is undecidable, so therefore it must not be RE.

Alternate Solution: Another way to show that E_{TM} is not RE is by reducing a known non-RE problem to it via a mapping reduction: we showed in class that if $A \leq_m B$ and A is not RE, neither is B (note that we proved this result only for mapping reductions: it's not true in general for Turing reductions). We can mapping-reduce $\overline{HALT}$ to E_{TM} as follows. Given $\langle M, x \rangle$, let N be the following TM:

1. Simulate M on x.
2. Accept.

Then if M does not halt on x, N will run forever on all inputs and so have empty language; otherwise it will accept all strings. So $\langle M, x \rangle \in \overline{HALT} \iff \langle N \rangle \in E_{TM}$, and therefore setting $f(\langle M, x \rangle) = \langle N \rangle$ gives a mapping reduction from $\overline{HALT}$ to E_{TM}. Therefore since $\overline{HALT}$ is not RE, neither is E_{TM}.

2. (40 pts.) A polynomial-time decidable problem

Let $L = \{\langle D, n\rangle \mid D \text{ is a DFA with } n \text{ states that accepts some string of length } n-1\}$. Prove that $L \in \mathsf{P}$.

Solution: We'll start by checking if D has n states, and rejecting if not (also rejecting if the input isn't of the form $\langle D, n\rangle$ at all). To see if D accepts some string of length $n-1$, we can't simply test all of these strings, since there are exponentially many of those (as long as the alphabet Σ of D has at least 2 symbols). Instead, we can check for the existence of a string in $L(D)$ of length $n-1$ by building a DFA accepting only those strings, and then checking whether it has empty language. To do this, first build a DFA A that accepts exactly the strings of length $n-1$: we can use a chain of $n+1$ states to keep track of how many symbols have been seen so far, from 0 up to n (once we get to n we don't need to keep counting, since we'll reject in any case). Then use the product construction on D and A to get a DFA P such that $L(P) = L(D) \cap L(A) = L(D) \cap \Sigma^{n-1}$. Finally, we can use the algorithm for E_{DFA} on P to check whether $L(P) = \emptyset$: if so, then $\langle D, n\rangle \notin L$, and otherwise $\langle D, n\rangle \in L$.

Since A has $n+1 = O(|D|)$ states and the same alphabet as D, its size is polynomial in $|D|$. So the product construction on D and A will also run in polynomial time, and P will have polynomial size. Therefore the algorithm for E_{DFA} will run in polynomial time too, and so the entire procedure will take polynomial time.

(*Note:* It's critical to the runtime here that D had n states, since the size of A was linear in n, not in the size of the binary encoding of n. If we allowed D to have any number of states, the algorithm would still work, but it could take exponential time: suppose D had only a single state, and we let n grow. The runtime will grow linearly with n, but the length of $\langle D, n\rangle$ will only grow *logarithmically* with n; so in terms of the length of its input, the algorithm will take exponential time.)

Alternate Solution: D accepts a string of length $n-1$ if and only if there is a path of that length from the start state to an accepting state. So we can use a variant of breadth-first search (BFS) to find all states which are reachable in exactly $n-1$ steps from the start state, and see if any of them are accepting states. Note that ordinary BFS will *not* work: BFS usually does not visit a vertex more than once, so that if for example some state q is reachable in 2 steps but *also* reachable in 5 steps along a different path, we will only visit q along the path of length 2 and fail to discover that it can also be reached along a path of length 5. To fix this problem, we'll allow a vertex to be visited multiple times, iteratively computing for $i = 0, 1, 2, \ldots n-1$ the set of states S_i reachable along a path of length i. We start out with $S_0 = \{q_0\}$, since only the initial state is reachable in 0 steps. Then having computed S_i, we can compute S_{i+1} by taking all states in S_i and finding all destinations of their outgoing transitions (formally, $S_{i+1} = \{\delta(q, s) \mid q \in S_i \text{ and } s \in \Sigma\}$). Once we have S_{n-1}, we accept if it contains any accepting state and otherwise reject.

Each iteration iterates over all states in S_i, of which there are at most $|D|$, and all the transitions from each such state, of which there are also at most $|D|$; so each iteration takes polynomial time. In total we need n iterations to compute $S_0, S_1, \ldots, S_{n-1}$, and as noted in the previous solution we have $n = O(|D|)$, so the overall time of the procedure is polynomial in the size of the input $\langle D, n\rangle$.

(*Note:* As in the previous solution, the runtime is linear in n, so the algorithm could take exponential time in the length of its input if we didn't require D to have n states.)

(*Note:* While these two solutions may seem quite different, they're basically doing the same thing under the hood. The states of the product automaton P in the first solution are of the form (q, i) where q is a state of D and $i \in \{0, \ldots, n\}$, and the start state is $(q_0, 0)$. For every transition from q to q' in D, there are corresponding transitions from (q, i) to $(q', i+1)$ in P for all $i < n$. So if the DFS performed by the E_{DFA} algorithm on P reaches state (q, i), that means that state q is reachable in D in exactly i steps from q_0. Therefore the reachable states in P for a given value of i are exactly the members of the set S_i in the second solution. Since the accepting states of P are of the form $(q, n-1)$ with q an accepting state of D, the DFS in the first solution finds a path to an accepting state if and only if S_{n-1} contains an accepting state.)

3. (60 pts.) Short answer

Answer each of the following questions, giving a 1-3 sentence justification.

Solution Note: I'm writing out my solutions in more detail than we asked for here, to help you better understand them.

(a) Consider the function $f : \{0,1\}^* \to \{0,1\}^*$ which converts binary to unary, i.e., if x is a binary representation of an integer n, then $f(x) = 1^n$ (so for example $f(101) = 11111$, $f(01) = 1$, and $f(000) = \varepsilon$). Is it possible to compute f in polynomial time?

Solution: No, f cannot be computed in polynomial time. This is because the output of f sometimes has to be exponentially-longer than its input: for example, a binary encoding of the integer $n = 2^m$ is the string $x = 10^m$, which has length $|x| = m+1$. But $f(x) = 1^n$, which has length $|f(x)| = n = 2^m$. So even writing the output of f on the tape in this case requires at least 2^m steps, which is not polynomial in the length of the input x. Therefore, f cannot be computed in polynomial time.

(b) What is wrong with the following argument? The language $L = 0^*1^*$ is regular, so it is in $\mathsf{TIME}(n)$ and thus in P. Now for any language $R \in \mathsf{P}$, there is a reduction $R \leq_P L$, so $R \in \mathsf{TIME}(n)$ as well. Therefore any problem solvable in polynomial time is solvable in linear time.

Solution: The only error in the argument is the inference that $R \in \mathsf{TIME}(n)$. A reduction $R \leq_P L$ means that an instance of R can be converted to an instance of L with the same answer in *polynomial time*, not necessarily linear time. The algorithm to solve R using this reduction would first apply the reduction to get an instance of L, then use the linear-time algorithm for L. The total time taken by this algorithm is not necessarily linear, since applying the reduction could take (say) n^2 or n^{10} time.

(*Note:* In particular, the claim that $R \leq_P L$ for all $R \in \mathsf{P}$ is correct: there are polynomial-time reductions between *any*[1] two problems in P. This is for the same reason there is a mapping reduction between any two decidable problems: the reduction can just *solve* the given problem and output an appropriate constant instance of the other problem based on whether the answer is yes or no.)

(c) Let $\mathsf{NP}_{\log}$ be a modified version of NP where the certificate for an input x is required to be of length $O(\log |x|)$. Argue that $\mathsf{P} = \mathsf{NP}_{\log}$.

Solution: Take any language $L \in \mathsf{NP}_{\log}$; since the length of the certificate for any input x is $O(\log |x|)$, it is $\leq c \log_2 |x|$ for some constant c (which could depend on L). But then there are at most $2^{c \log_2 |x|} = |x|^c$ possible values for the certificate, so we can go through all of them in polynomial time and for each one run the polynomial-time verifier for L to see if it accepts, accepting if so. This gives a polynomial-time algorithm for L, since if $x \in L$ then we will find a certificate value which the verifier accepts, and otherwise we will reject.

[1] Unless the right-hand side is $\emptyset$ or Σ^*, since then you wouldn't have anything to map yes/no answers to respectively, as we saw for mapping reductions.

4. (80 pts.) Scheduling is hard

Consider the following scheduling problem:

- There are students $S_1, \ldots S_n$, and classes $C_1, \ldots, C_m$ they can take.
- Each student has a different set of major requirements. There are three possible types of requirements:
  1. A requirement to take a specific class.
  2. An elective requirement, where the student must take at least one of a given set of classes.
  3. An incompatibility requirement, where a student is not allowed to take both of two given classes (e.g. if they are variants of the same class).

Given the students, classes, and requirements, the *schedule feasibility problem* is to determine whether it is possible to assign students to classes so that all requirements are satisfied. Let's prove that this problem is NP-complete.

(a) *(20 pts.)* Prove that the feasibility problem is in NP.

Solution: Given an instance of the problem, the certificate is just the assignment of students to classes: a verifier can then check whether every student satisfies their requirements. This can be done in polynomial time, since there are only linearly-many students, classes, and requirements, and checking a single requirement can clearly be done in polynomial time. So the feasibility problem is in NP.

(b) *(60 pts.)* Prove that the feasibility problem is NP-hard by constructing a polynomial-time reduction from *3SAT* to it.

(*Hint:* For each variable in the *3SAT* formula, create two classes: one where having a student in the class represents the variable being assigned true, and another representing the variable being assigned false. Then set up the class requirements so that an assignment of students to classes corresponds to a variable assignment that ensures at least one literal in each clause is assigned true.)

Solution: Suppose we have a 3SAT formula φ. Create a single student S, and classes C_x and $C_{\overline{x}}$ for each variable x in φ, which will represent the corresponding literals. We add an incompatibility requirement between C_x and $C_{\overline{x}}$ to prevent S from taking both classes, and an elective requirement saying it must take at least one. Then for each variable x the student must take exactly one of C_x and $C_{\overline{x}}$, representing whether we set x to true or false. Now to enforce the clauses, for each clause $c = \ell_1 \vee \ell_2 \vee \ell_3$ we add an elective requirement saying that S must take one of the three classes C_{ℓ_1}, C_{ℓ_2}, or C_{ℓ_3}. Then in any assignment of classes, at least one literal of c will be satisfied, and since this is true for each clause of φ, the assignment will satisfy φ. Conversely, if we have a satisfying assignment, the corresponding assignment of classes satisfies all our requirements. Therefore, our instance of the scheduling problem has a solution if and only if φ has a satisfying assignment. We can clearly construct the instance in polynomial time, since there are only polynomially-many students, classes, and requirements. So this gives a polynomial-time reduction from *3SAT* to the feasibility problem.

(*Note:* This shows that the problem is NP-hard even if you only allow a single student! On the other hand, the construction requires that student to take an arbitrarily-large number of classes, which isn't very realistic. If you generalize the problem to allow classes to have enrollment caps, then it remains NP-hard even if students only need to take 2 classes. More realistic versions of the problem with limited time slots, etc. would be at least as hard, so they would be NP-hard too.)

CSE 103 Computational Models
Fall 2021 Prof. Fremont HW 5

(3 questions, 160 points total)

1. (40 pts.) Designing a PDA

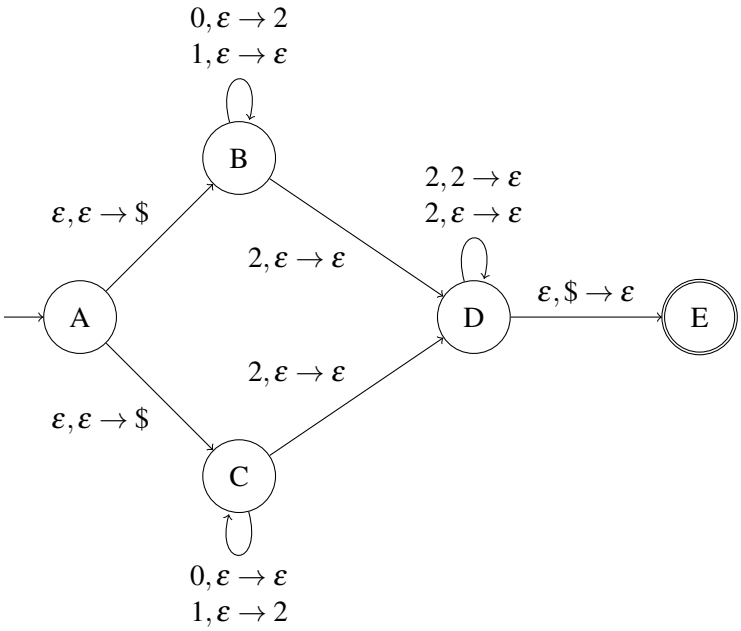
Given a binary string $x \in \{0,1\}^*$, let's write $Z(x)$ for the number of zeroes in x and $O(x)$ for the number of ones. Design a PDA with input alphabet $\Sigma = \{0,1,2\}$ recognizing the language

$$L = \{x2^k \mid x \in \{0,1\}^* \text{ and } k > \min(Z(x), O(x))\};$$

that is, strings consisting of a binary string followed by some number of 2s, such that 2 is not the least-common digit overall (for example, $2, 12, 00022, 01022 \in L$, whereas $012, 21 \notin L$).

(*Hint:* Remember that PDAs can be nondeterministic.)

Solution: If we only had to compare the number of 2s against the number of 0s, we could use the approach we've seen before: push the 0s onto the stack, and then pop one whenever we see a 2, making sure we have enough 2s to completely clear the stack (plus one extra 2, since we need to have strictly more 2s than 0s). However, this approach won't work to keep track of *both* the number of 0s and the number of 1s we've seen, since we only have one stack and must decide whether to pair 2s with 0s or with 1s. The key idea is that since PDAs can be nondeterministic, we can *guess* which number to pair against: since $k > \min(Z(x), O(x))$ if and only if either $k > Z(x)$ or $k > O(x)$, we can build separate PDAs for each of the latter conditions and then nondeterministically choose which one to use. Combining some common parts of the two PDAs, this gives us:



In state A, we can choose to go to either state B or state C, pushing a \$ on the stack so that we can detect an empty stack later. In state B, we push a 2 on the stack for every 0 we see, ignoring any 1s; state C does likewise with 0 and 1 reversed. When we see a 2, we move to state D, whereupon we can not have any more

0s or 1s, but we can read 2s, with the (nondeterministic) option to pop a 2 off the stack for each one. If we clear the stack, so that \$ is on top, we can transition to state E and accept if there are no more input symbols.

(*Note:* There are many slightly-different ways of setting up the PDA that work just as well: for example, we could have moved the $2, \varepsilon \to \varepsilon$ self-loop at state D to state E; we could also have pushed 0s and 1s onto the stack unchanged rather than converting them to 2s, as long as we had transitions allowing us to pop either of them when we read a 2.)

If $w \in L$, then it is of the form $x2^k$ with $x \in \{0,1\}^*$ and either $k > Z(x)$ or $k > O(x)$. In the first case, if we transition from A to B, after reading x the stack will have $Z(x)$ copies of 2 on it; we can then read in $1 + Z(x) \leq k$ copies of 2 from the input using the transition from B to D and the self-loop at D. After this, the stack will only have a \$ on it; if there are any further copies of 2 in x we can read them using the $2, \varepsilon \to \varepsilon$ self-loop before transitioning to E and accepting. If instead $k > O(x)$, we can transition from A to C and build an accepting path in an exactly analogous way to above, just exchanging 0s and 1s. So in either case our PDA will accept w.

Conversely, if our PDA accepts a word $w \in \Sigma$, then it must reach state D with an empty stack (other than the \$). If the accepting path went through state B, then upon entering state D for the first time, the stack will contain $Z(x)$ copies of 2, and since the only transition that pops a symbol in state D is $2, 2 \to \varepsilon$, the input must contain at least $Z(x) + 1$ copies of 2 (where the $+1$ comes from the transition from B to D). Likewise, if the accepting path went through state C, the input must contain at least $O(x) + 1$ copies of 2. Furthermore, any 2s in the input must come after the 0s and 1s, since only transitions leading to state D can read a 2. So w must be of the form $x2^k$ where $x \in \{0,1\}^*$ and $k > \min(Z(x), O(x))$.

Therefore, the language of our PDA is L, as desired.

2. (60 pts.) Closure properties of context-free languages

Recall that we proved that the regular languages are closed under various operations, including union, concatenation, Kleene star, intersection, and complement. Let's see what happens when these operations are applied to context-free languages.

(a) Suppose we have two CFGs G_1 and G_2, with start variables S_1 and S_2 respectively. What rule(s) should we add to get a grammar (say with start variable S) for $L(G_1) \cup L(G_2)$?

Solution: We should add the rule $S \to S_1 \mid S_2$. Then we can derive any $w \in L(G_1)$ by first applying the rule $S \to S_1$ and then proceeding according to the derivation of w in G_1, and similarly for any $w \in L(G_2)$. Conversely, any derivation in G must start with either the rule $S \to S_1$ or $S \to S_2$ (since these are the only rules with S on the left-hand side), so any string in $L(G)$ is a member of either $L(G_1)$ or $L(G_2)$ (or both).

(*Note:* This argument assumes that G_1 and G_2 have no variables in common: otherwise, you might be able to derive something from S_1 using some of the rules from G_2, which would then not necessarily be in either $L(G_1)$ or $L(G_2)$. We can always make sure that the variables of G_1 and G_2 are disjoint by renaming variables as needed.)

(b) If instead we want a grammar for $L(G_1)L(G_2)$ (the concatenation of any word in $L(G_1)$ with any word in $L(G_2)$), what rule(s) should we add?

Solution: We add the rule $S \to S_1 S_2$. An argument along the lines of that in part (a) then shows that $L(G)$ consists exactly of those strings which are concatenations of a word in $L(G_1)$ and a word in $L(G_2)$.

(c) How about the Kleene star $L(G_1)^*$?

Solution: We add the rule $S \to SS_1 \mid \varepsilon$. Then we can apply the rule $S \to SS_1$ repeatedly $n \geq 0$ times to derive the string SS_1^n, then apply $S \to \varepsilon$ to get S_1^n, and finally expand each copy of S_1 into a word of $L(G_1)$. So $L(G_1)^n \subseteq L(G)$ for every $n \geq 0$, and so $L(G)^* = \cup_{n \geq 0} L(G_1)^n \subseteq L(G)$. Conversely, any $w \in L(G)$ is either ε or a concatenation of a finite number of words from $L(G_1)$, so $L(G) \subseteq L(G_1)^*$.

(d) The previous three parts show that the context-free languages are closed under the regular operations (union, concatenation, and Kleene star). Prove that the context-free languages are *not* closed under intersection, i.e., there are context-free languages L_1 and L_2 such that $L_1 \cap L_2$ is not context-free.

(*Hint:* Start by arguing that the languages $\{0^n 1^n 2^* \mid n \geq 0\}$ and $\{0^* 1^n 2^n \mid n \geq 0\}$ *are* context-free.)

Solution: We gave a grammar for $0^n 1^n$ in class, and 2^* is context-free since it is regular, so by closure under concatenation (from part (b)) so is L_1. Alternatively, we can write down a grammar for it directly:

$$\begin{aligned} S &\to AB \\ A &\to 0A1 \mid \varepsilon \\ B &\to 2B \mid \varepsilon \end{aligned}$$

Similar arguments show that L_2 is also context-free.

Now if the context-free languages were closed under intersection, the language $L_1 \cap L_2 = \{0^n 1^n 2^n \mid n \geq 0\}$ would be context-free, but we proved in class that it is not. So the context-free languages are not closed under intersection.

(e) Prove that the context-free languages are not closed under complement.

(*Hint:* Assume that CFLs *are* closed under complement, and get a contradiction with part (d).)

Solution: For any context-free languages L_1 and L_2 we have $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ (in fact this is true for any two languages). If the context-free languages were closed under complement, then $\overline{L_1}$ and $\overline{L_2}$ would also be context-free, and since CFLs are closed under union as we saw in part (a), $\overline{L_1} \cup \overline{L_2}$ would

be context-free. Applying closure under complement again, $\overline{\overline{L_1} \cup \overline{L_2}}$ would be context-free, but this is the same as $L_1 \cap L_2$, and therefore the intersection of any two CFLs would be context-free. This contradicts part (d).

3. (60 pts.) The pumping lemma for context-free languages

Let's prove that the language $L = \{0^i 1^j 2^k \mid i, j, k \geq 0 \text{ and } j < i, k\}$ is not context-free.

(a) Given a pumping length $p \geq 1$, choose an appropriate word $w \in L$ with $|w| \geq p$.

Solution: One possible word is $w = 0^{p+1}1^p 2^{p+1}$. As required, we have $w \in L$ and $|w| = 3p+2 \geq p$.

(b) Given a decomposition $w = uvxyz$ with $|vy| > 0$ and $|vxy| \leq p$, we need to pick a $k \geq 0$ such that $uv^k xy^k z \notin L$. Show how to do this in the case when vy contains a 1.

(*Hint:* Can vy also contain both a 0 and a 2?)

Solution: Since $|vxy| \leq p$, it is not possible for vxy to contain all three symbols 0, 1, and 2, since the 0s and 2s in w are separated by p copies of 1. So vy either contains no 0s or no 2s (or both): let us call (one of) the missing symbols m. If we put $k = 2$, then uv^2xy^2z will contain strictly more 1s than $uvxyz = w$, since vy contains a 1 by assumption, but it will contain the same number of ms. So uv^2xy^2z will have at least $p+1$ 1s but exactly $p+1$ ms, and therefore will not be in L.

(c) Show how to pick $k \geq 0$ so that $uv^k xy^k z \notin L$ in the other case, namely when vy contains no 1s.

Solution: If vy contains no 1s, then since $|vxy| \leq p$, either vy consists only of 0s or only of 2s: we can't have both, since the 0s and 2s are separated by p copies of 1. Say that vy is made up of ms. Then picking $k = 0$, the string $uv^k xy^k z = uxz$ contains strictly fewer ms than w, since $|vy| > 0$, and the same number of the other two symbols. But then uxz has at most p copies of m, while it has exactly p copies of 1, and therefore will not be in L.