Problems marked with **E** are graded on effort, which means that they are graded subjectively on the perceived effort you put into them, rather than on correctness. For bonus questions, we will not provide any insight during office hours or Piazza, and we do not guarantee anything about the difficulty of these questions. We strongly encourage you to typeset your solutions in LaTeX.

If you collaborated with someone, you must state their names. You must write your own solution and may not look at any other student's write-up.

0. If applicable, state the name(s) and uniqname(s) of your collaborator(s).

**Solution:**

1. *Alphabets, Languages, and Decision Problems.*

**E** (a) In your own words, define the following terms and explain the relationship between them: *input alphabet*, *tape alphabet*, *language*, *decider*.

**Solution:** The input alphabet is the set of characters (example: {0, 1}) that can be input into a Turing Machine's tape. It is a subset of the tape alphabet.

The tape alphabet is the set of characters (example, {0, 1, $\perp$}) that can be written on a Turing Machine's tape by the machine itself or appear there by default as a "blank" character. It is a superset of the input alphabet.

A language is a set of elements that share a common property (example: the set of all binary strings that have an even length). Every element in the language is made up of characters from an (input) alphabet. Any arbitrary language is defined over a corresponding alphabet.

A decider is defined as the Turing Machine $M$ that decides a language $L$. $M$ is a program that accepts an input, and determines whether or not that input is in the language $L$. The program $M$ must accept all inputs that are in $L$ and reject all inputs that are not in $L$. The program $M$ must also halt on all inputs (no infinite loops!)

For part (b), define a language $L$ which corresponds to each decision problem stated in parts i through iii. **Example:** Given integers $x,\ y$, determine if $x$ is divisible by $y$.

$$L = \{\langle x \in \mathbb{Z},\ y \in \mathbb{Z} \rangle : x \bmod y \equiv 0\}$$

(b) i. Given an integer $x$, is $x$ a perfect cube?

**Solution:** We define

$$L = \{x \in \mathbb{Z} : x = n^3 \text{ for some } n \in \mathbb{Z}\}.$$

The language is limited to such representations of numbers that are perfect cubes, because, as we can see, this Turing Machine will only accept an input $x$ where it is equal to a $n^3$ where $n$ is also an integer.

ii. Given an integer $m$, is $m$ odd?

**Solution:**

$$L = \{m \in \mathbb{Z} : m \bmod 2 \equiv 1\}$$

We must check that the integer $m$ is divisible by 2 to check if it is an odd number. In this case, a Turing Machine would only accept a $m$ where if this $m$ is divided by 2, there would be a remainder of one, meaning that it is an odd number by definition.

iii. Given natural numbers $p$ and $q$, do $p$ and $q$ share a common prime factor?

**Solution:**

$$L = \{\langle p \in \mathbb{N},\ q \in \mathbb{N} \rangle : \gcd(p, q) \geq 2\}.$$

If $p$ and $q$ share a common prime factor, then their greatest common factor must be greater than 1. Therefore, we must check if $\gcd(p, q) \geq 2$.

2. *Deciders.* Consider the language:

$$L = \{\langle G = (V, E),\ v_1,\ v_2,\ k\rangle : \text{There exists a path } P \subseteq E \text{ between } v_1 \text{ and } v_2 \text{ s.t. } |P| \leq k\}$$

where $G$ is a finite graph represented by the vertex set $V$ and the edge set $E$, $v_1$ and $v_2$ are distinct vertices in $V$, and $k \in \mathbb{N}$ is the maximum allowed path length in terms of number of edges.

To a computer, everything is bits. The notation $\langle G = (V, E), v_1 \in V, v_2 \in V, k\rangle$ means the 1's and 0's that represent the variables $G$, $v_1$,$v_2$, and $k$. Thus, we are able to use $G$, $v_1$, $v_2$, and $k$ as inputs to some program.

(a) Is $L$ a decidable language? If $L$ is decidable, prove your answer by writing the pseudocode for a program $M$ that decides the language $L$, otherwise, justify why $L$ is undecidable.

**Solution:** $L$ is decidable because there is a decider for $L$. We describe several possible deciders below.

There exists a path between two vertices in the graph if they are reachable by traversing the graph from one of the two vertices to the other.

If one of the endpoint vertices is visited during a traversal started from the other endpoint within $k$ iterations, then there exists a path with length $\leq P$ between those vertices. However, if one of the endpoint vertices is not visited during a traversal started from the other endpoint within $k$ iterations, then there either does not exist a path between those vertices, or if there exists one, its length is greater than $k$.

The two most common approaches to graph traversal are depth-first search and breadth-first search. The iterative implementations to these algorithms are included below. There are equivalent recursive implementations as well.

These approaches are not the only way to decide the language. A correct algorithm just needs to traverse the entire graph to search for the path.

**Depth-first Search Approach (Iterative):**
$M$ = "On input $(G = (V, E), v_1, v_2, k)$ :

1) let $S$ be a stack
2) start at vertex $v_1$ by pushing $v_1$ onto $S$.
3) Initialize all vertices as "not visited"
4) Initialize $v_1$'s distance as 0
5) While $S$ is not empty:
    a) pop a vertex $v$ off the top of $S$
    b) If $v = v_2$, accept
    c) Mark $v$ as visited
    d) If $v$'s distance $< k$, push all of $v$'s adjacent non-visited vertices to the top of $S$, and set their "distance" to $v$'s distance plus one
6) reject

**Breadth-first Search Approach (Iterative):**
$M$ = "On input $(G = (V, E))$ :

1) let $Q$ be a queue
2) start at vertex $v_1$ by pushing $v_1$ onto $Q$.
3) Initialize all vertices as "not visited"
4) Initialize $v_1$'s distance as 0
5) While $Q$ is not empty:
    a) pop a vertex $v$ from the front of $Q$
    b) If $v = v_2$, accept
    c) Mark $v$ as visited
    d) If $v$'s distance $< k$, push all of $v$'s adjacent non-visited vertices to the end of $Q$, and set their "distance" to $v$'s distance plus one
6) reject

Both approaches traverse all nodes that are within $k$ edges of $v_1$, accept if $v_2$ is found, and reject if $v_2$ is not found (i.e. the stack/queue is emptied) during traversal.

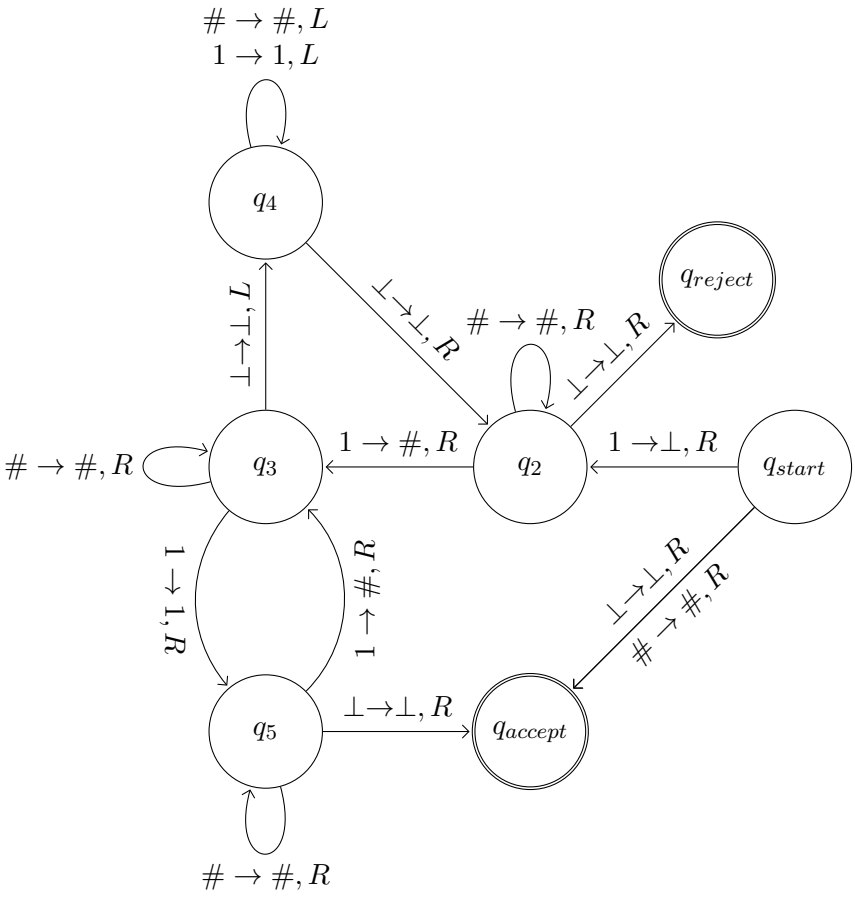
(b) Assuming your program $M$ is a decider for the language $L$, are there any inputs to $M$ that would cause it to loop forever? If so, give one example of an input that causes infinite looping. Otherwise, explain why no such inputs occur.

**Solution:** No such inputs exist. Since $M$ decides the language $L$, $M$ must halt on all inputs by definition of a decider.

(c) Prove that the set of all deciders is countable.
**Hint:** First prove that the set of all Turing machines is countable.

**Solution:** Every Turing machine can be encoded in a finite-length bitstring. The set of all finite-length bitstrings are countable; you can create a bijective function with the natural numbers such as $\epsilon \to 0$, $0 \to 1$, $1 \to 2$, $00 \to 3$, $01 \to 4$, $10 \to 5$, and so on. Therefore, the set of all Turing machines is countable. Deciders form a subset of Turing machines, so the set of deciders must also be countable.

3. *Turing Machines.* Consider the Turing machine represented by the following figure, for input alphabet $\Sigma = \{1\}$ and tape alphabet $\Gamma = \{1, \#, \bot\}$. (Note that there are *two* extra characters in the latter.)




(a) Write out the seven tuple definition of the above Turing Machine.

**Solution:** The seven tuple definition of the Turing Machine is:

$$TM = \langle Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}} \rangle$$

where $Q$ is the set of all states,

$$q_2, q_3, q_4, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}}$$

$\Sigma$ is the input alphabet,

$$\Sigma = \{1\}$$

$\Gamma$ is the tape alphabet,

$$\Gamma = \{1, \#, \bot\}$$

$\delta$ is the transition function,

$$\delta(q_{start}, 1) = (q_2, \bot, R)$$

$$\delta(q_{start}, \#) = (q_{accept}, \perp, R)$$
$$\delta(q_{start}, \perp) = (q_{accept}, \perp, R)$$

$$\delta(q_2, 1, ) = (q_3, \#, R)$$
$$\delta(q_2, \#) = (q_2, \#, R)$$
$$\delta(q_2, \perp) = (q_{reject}, \perp, R)$$

$$\delta(q_3, 1) = (q_5, 1, R)$$
$$\delta(q_3, \#) = (q_3, \#, R)$$
$$\delta(q_3, \perp) = (q_4, \perp, L)$$

$$\delta(q_4, 1) = (q_4, 1, L)$$
$$\delta(q_4, \#) = (q_4, \#, L)$$
$$\delta(q_4, \perp) = (q_2, \perp, R)$$

$$\delta(q_5, 1) = (q_3, \#, R)$$
$$\delta(q_5, \#) = (q_5, \#, R)$$
$$\delta(q_5, \perp) = (q_{accept}, \perp, R)$$

(b) What language does this Turing machine decide (if any)? Justify your answer.

**Solution:** This Turing machine decides the language:

$$\{x \mid x \notin \{1\}^{2^n} \text{ for } n \in \mathbb{N}_0\}$$

We will consider some example runs of the machine to understand how this Turing machine works, and argue how this example generalizes to any string in $\{1\}^*$.

Consider the input string (which is in the above language) "11111111" and note that the rest of the tape is filled with $\perp$:

- First we will have $q_{start}$ and transition to $q_2$ when we detect the first 1 giving us the string: "$\perp$ 1111111"
- We detect a 1 so we transition to state $q_3$ as follows: "$\perp$ #111111"
- We now do a loop making every other 1 a # until we hit the end of the string, the result is as follows: "$\perp$ #1#1#1#"
- We then move to start $q_4$ reversing directions now and moving left. The function of state $q_4$ is to take us back to the beginning of the string (the first $\perp$ symbol). After we hit the first $\perp$ symbol we are at $q_2$ with the following on our tape: "$\perp$ #1#1#1#"

- We repeat the process again replacing every other 1 with a # and when we reach the end of the string we have:

  "$\perp$ ###1###"

- We reset back to the beginning of the tape and are again at:

  "$\perp$ ###1###"

- After another iteration we are at the end of the tape with:

  "$\perp$ #######"

- We skip ever going to $q_5$ this time since we are already at the end of the tape after one replacement. We transition back to the start to state $q_2$:

  "$\perp$ #######"

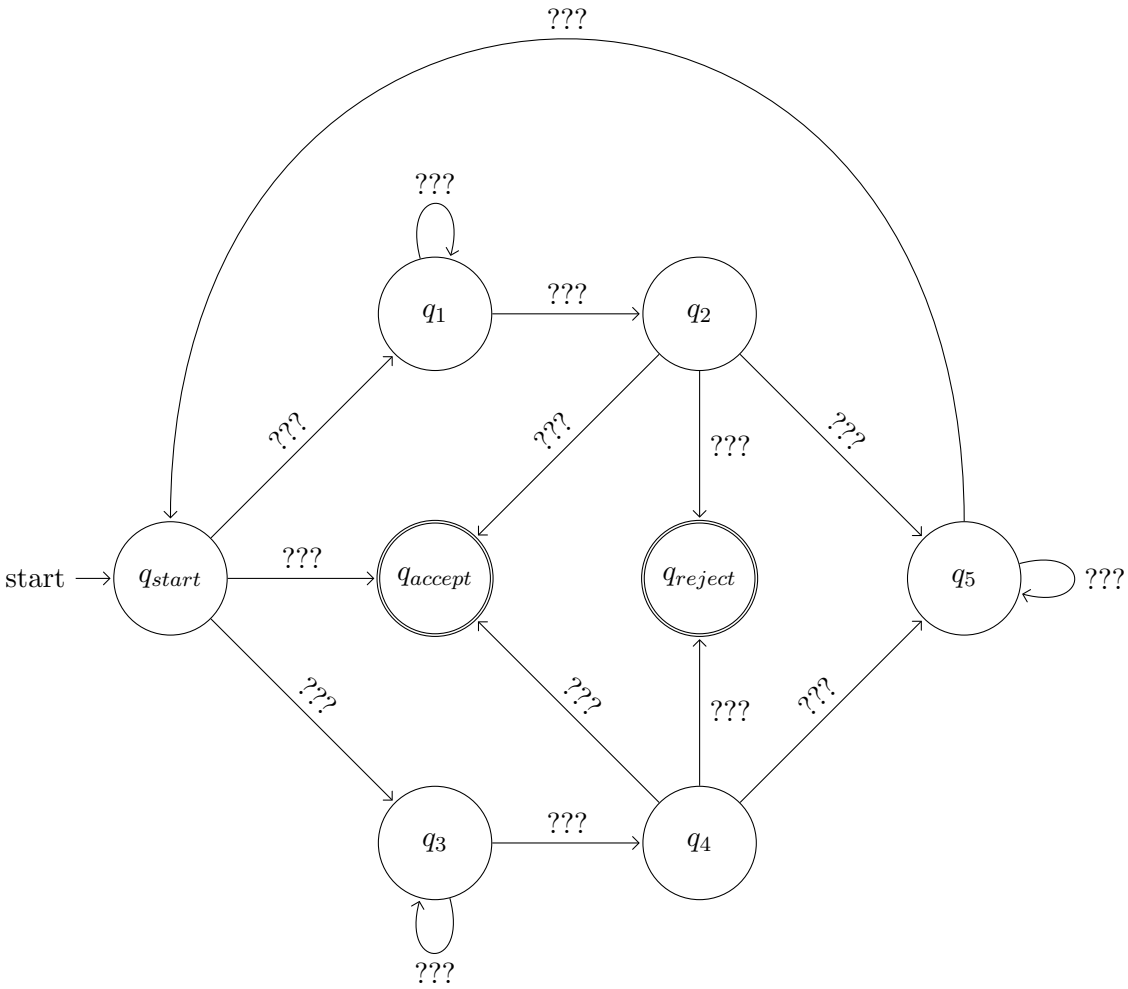
- At $q_2$ we loop through the rest of the string until we hit a $\perp$ since it is composed entirely of # characters now. We notice that if we ever hit a $\perp$ symbol while at state $q_2$ we transition to state $q_{reject}$, and thus we have finished, rejecting the string $\{1\}^{2^3}$ as expected.

More generally, each pass of the machine over the input replaces every other 1 with #, and accepts if the number of 1s it encounters is an odd number greater than one. Therefore, the machine rejects if and only if every pass sees an even number of 1s, followed by a final pass with exactly one 1. The machine therefore rejects every string of the form $x \in \{1\}^{2^n}$, and accepts all others.

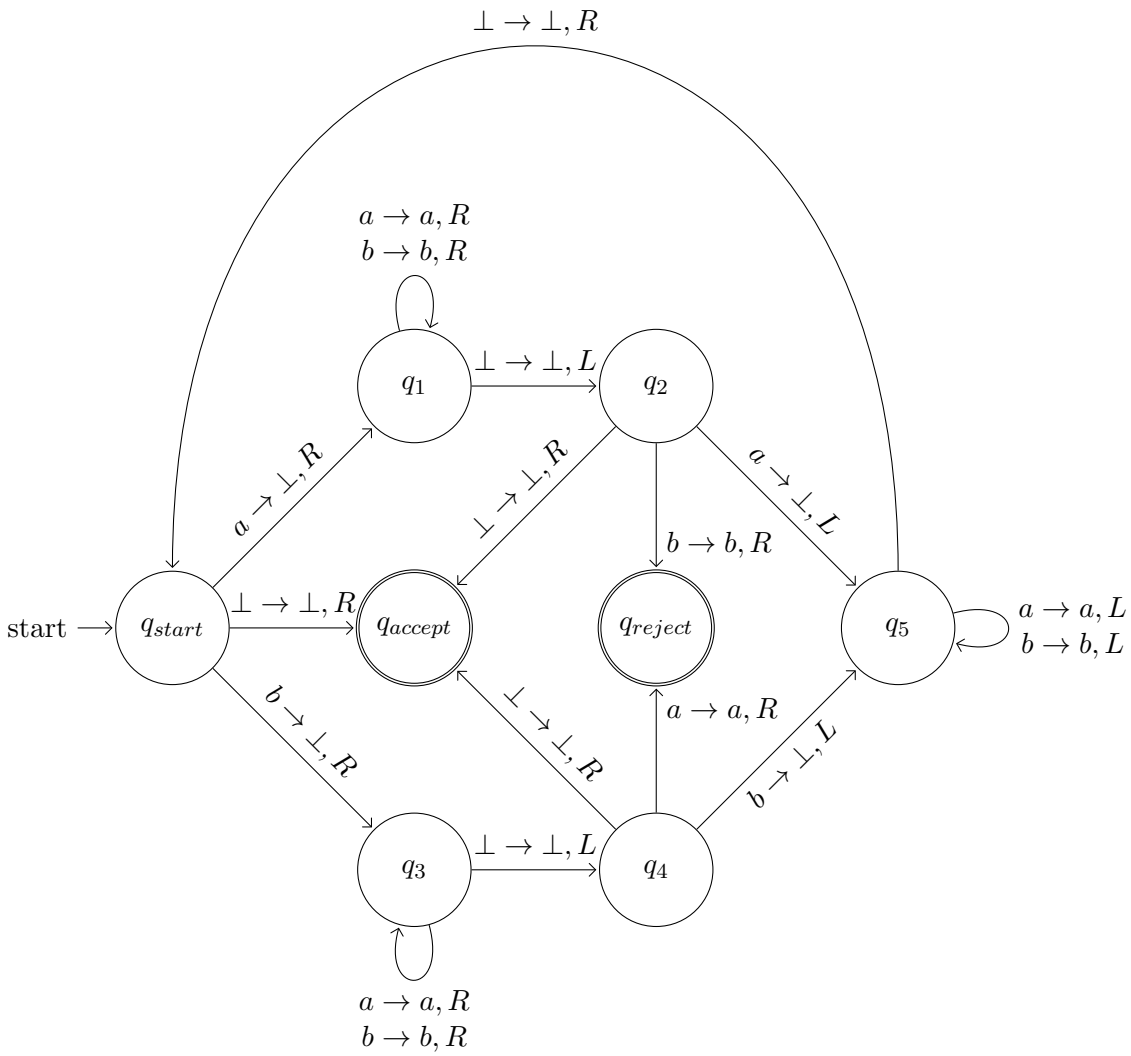
4. *Turing Machines.* A string $x$ is a *palindrome* if $x = x^R$, where $x^R$ denotes the reversal of $x$ (i.e., the characters of $x$ in reverse order). For example, `abba` and `ababa` are palindromes, and `abbab` and `aaba` are not palindromes.

   The state diagram below is for a Turing machine that decides the language of palindromes over the alphabet $\Sigma = \{a, b\}$. **Complete the diagram by supplying the appropriate transitions** of the form "$e \to e', L/R$" where $e, e'$ are symbols from the tape alphabet $\Gamma = \{a, b, \perp\}$. As a hint, each of the eight states serves one of the following purposes:

   - The initial, accept, and reject states.
   - After seeing an $a$ (respectively, $b$) at the left end of the (remaining) input string, move the head to the right end of the (remaining) string.
   - After seeing an $a$ (respectively, $b$) at the left end, the head is at the right end.
   - Move the head to the left end of the (remaining) input string.



**Solution:**


$\perp \to \perp, R$
$a \to a, R$
$b \to b, R$
$q_1$
$\perp \to \perp, L$
$q_2$
$a \to \perp, R$
$\perp \to \perp, R$
$b \to b, R$
$a \to \perp, L$
start
$q_{start}$
$\perp \to \perp, R$
$q_{accept}$
$q_{reject}$
$q_5$
$a \to a, L$
$b \to b, L$
$b \to \perp, R$
$\perp \to \perp, R$
$a \to a, R$
$b \to \perp, L$
$q_3$
$\perp \to \perp, L$
$q_4$
$a \to a, R$
$b \to b, R$

**E** 5. *Turing Machine Equivalence.* A *modify-twice* Turing Machine is a Turing Machine which can modify each cell of the tape at most twice. Formally, it is a 7-tuple $(Q, \Sigma, \Gamma \times \{0,1,2\}, q_0, q_{accept}, q_{reject}, \delta)$, where

$$\begin{aligned} Q &\text{ is the set of states, including } q_0, q_{\text{accept}}, q_{\text{reject}}, \\ \Sigma &\text{ is the input alphabet,} \\ \Gamma \times \{0,1,2\} &\text{ is the tape alphabet (see below),} \\ q_0 &\text{ is the start state,} \\ q_{accept} &\text{ is the accept state,} \\ q_{reject} &\text{ is the reject state,} \\ \delta\colon (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R\} &\text{ is the transition function.} \end{aligned}$$

As compared with a regular Turing Machine, we modify the tape alphabet so every cell of the tape stores $(\gamma, n)$ for some $\gamma \in \Gamma, n \in \{0,1,2\}$. The character $\gamma$ is the symbol that would appear on the tape of a normal Turing machine. The integer $n$ is the cell's "count," i.e., the number of times that the machine has written to the cell.

Initially, the tape contains the input string and blanks as with a normal Turing Machine, with all counts set to 0. Each computation step is just as with a normal Turing Machine (note that the transition function only "sees" the $\Gamma$-symbol stored at the current cell, not the count), but when the machine modifies a cell (i.e. if the symbol on the cell changes), its count is automatically incremented. If the machine ever attempts to modify a cell whose count is already 2, the computation aborts and the result of the computation is *undefined* (neither accept, nor reject, nor loop).

Show that modify-twice Turing Machines are equivalent in power to regular Turing Machines. That is, if some regular Turing Machine decides a language $L$, then so does some modify-twice Turing Machine, and vice-versa.

**Solution:**

First, we must prove that a regular TM can simulate a modify-twice TM. Since the only difference is that a regular TM can modify a cell an arbitrary number of times, while a modify-twice TM can do so only twice, any valid computation on a modify-twice TM is also valid on a regular TM using the same 7-tuple but with tape alphabet $\Gamma$ instead of $\Gamma \times \{0,1,2\}$.

Now, we must prove that a modify-twice TM can simulate a regular TM. The modify-twice TM can simulate the original TM by repeatedly copying the entire active portion of the tape over to a fresh section of the tape to the right. A marker (e.g. # ) is used to separate the new portion of tape and the old portion of tape. The modify-twice TM copies the active portion of the tape each time it executes a single step of the original TM.

For a regular TM with tape alphabet $\Gamma$, the equivalent modify-twice TM has tape alphabet $\Gamma \cup \{\#\} \cup \{\dot{\gamma} : \gamma \in \Gamma\}$ (excluding the modify count). Start by placing a # to the right of the input and copying the entire input to the right of that #, with another # to the right. The first input symbol is replaced with a dotted version of the symbol. Each time

a symbol is copied, it is replaced with # as a marker to indicate what we have already copied. So if the original input is 01101, we end up with:

$$\#\#\#\#\#\#\dot{0}1101\#$$

The contents between the hash symbols are the active part of the tape. Then move the head to the current input symbol, which is marked by the dot. Now apply the transition function of the original TM. Keep track of the new symbol and direction as part of the simulating TM's state, but do not actually modify the symbols on the tape. Instead, copy the entire active part of the tape over to the right of the last hash, but replace the dotted symbol with its new value and its neighbor to the left or right with the dotted version according to whether the head was to move left or right. Each time we copy a symbol, we replace it with a # to indicate it was copied. For instance, if the new symbol in the example above is 1 and the head moves to the right, we end up with:

$$\#\#\#\#\#\#\#\#\#\#\#1\dot{1}101\#$$

Observe that we use a fresh part of the tape for each step, and each cell gets modified twice – first from the initial $\perp$ to the contents of the respective cell (with a dot to mark the head location), then to # when we copy the cell to a new location after the next step.

One last detail is that if the head is on the last symbol (to the left of the hash on the right) and then moves to the right, we expand the active portion of the tape by a blank symbol when we copy it over for the next step. Similarly, if the head is on the first active symbol (just to the right of the string of hashes) and the transition function indicates a move to the left, we do not move the dotted head location in the simulation – this matches the behavior of a regular TM staying in the first cell when the head is supposed to move left.

6. *Tree Graphs.* In lecture, three equivalent definitions for a tree graph were given. Let $G = (V, E)$ be an undirected graph. Prove that the following two definitions (from lecture) for a tree graph $G$ are equivalent:

   (a) $G$ is connected without cycles.

   (b) $G$ is maximal without cycles.

**Solution:** In this problem, we will consider graphs with more than two vertices, since smaller graphs cannot contain cycles and so are trivially maximal acyclic if they are connected and vice versa.

First we want to prove (a) $\implies$ (b).

Let $G$ be a connected, acyclic graph. Since $G$ is acyclic, there exist two vertices $u$ and $v$ that do not have an edge between them. Since $G$ is connected, there exists a path $P$ that connects $u$ and $v$. Now suppose we draw an edge $e$ between $u$ and $v$. This would induce a cycle in $G$, namely $P \cup e$. Since we cannot add an edge in $G$ without inducing a cycle, $G$ is maximal. Thus, (a) $\implies$ (b).

Now we show that (b) $\implies$ (a).

Let $G$ be a maximal acyclic graph. In order to show that $G$ is connected, we will show that any pair of vertices have a path between them. Let $u, v$ be any two vertices in $G$.

**Case 1**: There is an edge $e$ between $u$ and $v$. If this is true, then there is trivially a path between $u$ and $v$, namely this edge.

**Case 2**: There is not an edge between $u$ and $v$. Suppose we draw an edge $e$ between $u$ and $v$. Since $G$ is maximally acyclic, this would induce a cycle in $G$. Thus, there must be a path from $u$ to $v$ not including $e$.

Since there is a path between any two vertices in $G$, $G$ is connected.

Therefore, because we have now shown that (b) $\implies$ (a) and (a) $\implies$ (b), we have proved their equivalence.

# Optional bonus questions

For bonus questions, we will not provide any insight during office hours or Piazza, and we do not guarantee anything about the difficulty of these questions. *Only attempt these questions **after** you have finished the rest of the homework.*

1. *Computablity.* In 1936, Alan Turing's ground breaking "On Computable Numbers, With An Application To The Entscheindungsproblem" was published. It begins with:

   "The 'computable' numbers may be described briefly as the real number whose expressions as a decimal are calculable by a finite means."

   Turing expands upon the above by describing, what is now known as, the Turing machine. More precisely: A computable number is one for which there is a Turing machine that when given $n$ on its initial tape, terminates with the $n$-th digit of that number encoded on its tape.

   Show that $\sqrt{376}$ is a computable number.

   *Hint: C++, Python, P", JavaScript etc. are all Turing complete*

**Solution:**

Although it is irrational, we can compute $\sqrt{376}$ to an arbitrary precision of digits. The following Python program uses Newton's method to compute $\sqrt{376}$ to an arbitrary precision $p$:

```
def sqrt376(p):
    x = 1
    while p > 0:
        x -= (x * x - 376) / (2 * x)
        p = p - 1
    return x
```

As $p$ increases, the number of decimal places corresponding with $\sqrt{376}$ also increases. Although the above program uses Python, we could write a program in any Turing complete language, or create a Turing machine itself, that has the same functionality. Since a Turing machine gives us the power to compute $\sqrt{376}$ to an arbitrary precision, $\sqrt{376}$ is a computable number despite being irrational.