(a) No, $L$ is not context-free. Suppose to the contrary that it is, and let $p$ be the pumping length of $L$. Let $w = s\#2^p$, where $s$ is the binary representation of the number $n = 3^p - 1$, without any leading zeros. So $|w| = \lfloor \log_2 n \rfloor + 1 = O(p)$. By the pumping lemma we can write $w = uvxyz$, such that $|vy| \leq p$, and $vy \neq \epsilon$. If $vy$ contains the # then $uv^ixyz \notin L$ contains $i$ copies of # and so is only in $L$ when $i = 1$. If $vy$ lies entirely after the #, then in $uv^2xy^2z$ the portion before the # represents $n$ while the portion after represents a number larger than $n$. So it is not in $L$. The same argument applies when $vy$ lies entirely before the #. In the remaining cases $vxy$ straddles the # (i.e., $x$ must contain #). So $|v| + |y| < p$. Because we are not in one of the earlier cases, $|v| \neq 0 \neq |y|$, and $v$ is in $s$, and $y$ is in $2^p$. Consider $uv^ixy^iz$ for $i \geq 2$. So the binary representation can represent a number that is at most $2^{|x| + (i-1)|y|} - 1$, but the ternary representation represents the number $3^{n+|y|} = 3^{p+|y|}$. Since the second number grows faster than the first number as a function of $i$, for some value of $i$, the number represented by the ternary representation must be bigger than the number represented by the binary representation, i.e. there exists an $i$ such that $uv^ixy^iz \notin L$ – contradiction.

(b) The input is formatted as $b\#t$ with $b$ a binary representation of a number $m$ and $t$ a ternary representation of a number $n$. We are to accept if and only if $m = n$ (numerically). When $b = \varepsilon$, $m = 0$; when $t = \varepsilon$, $n = 0$. Leading zeroes on $b$ and $t$ are allowed.

**Idea:** $m = n$ if and only if both are 0 or $m - 1 = n - 1$. So keep decrementing both until one becomes 0, and then check that the other one is now 0.

<table>
<thead>
<tr>
<th>State ↓ Symbol →</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>#</th>
<th>↓</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>(B, ⊥, R)</td>
<td>(C, ⊥, R)</td>
<td>(E, ⊥, R)</td>
<td>(F, 0, L)</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>(B, 0, R)</td>
<td>(C, 0, R)</td>
<td>(D, 0, R)</td>
<td>(E, 0, R)</td>
<td>(F, 0, L)</td>
</tr>
<tr>
<td>C</td>
<td>(B, 1, R)</td>
<td>(C, 1, R)</td>
<td>(D, 1, R)</td>
<td>(E, 1, R)</td>
<td>(H, 0, L)</td>
</tr>
<tr>
<td>D</td>
<td>(B, 2, R)</td>
<td>(C, 2, R)</td>
<td>(D, 2, R)</td>
<td>(E, 2, R)</td>
<td>(H, 1, L)</td>
</tr>
<tr>
<td>E</td>
<td>(B, #, R)</td>
<td>(C, #, R)</td>
<td>(D, #, R)</td>
<td>(E, #, R)</td>
<td>(M, #, L)</td>
</tr>
<tr>
<td>F</td>
<td>(F, 0, L)</td>
<td>(G, 0, R)</td>
<td>(G, 1, R)</td>
<td>(M, #, L)</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>(G, 2, R)</td>
<td></td>
<td></td>
<td></td>
<td>(H, ⊥, L)</td>
</tr>
<tr>
<td>H</td>
<td>(H, 0, L)</td>
<td>(H, 1, L)</td>
<td>(H, 2, L)</td>
<td>(I, #, L)</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>(I, 0, L)</td>
<td>(J, 0, R)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>(J, 1, R)</td>
<td></td>
<td></td>
<td>(K, #, R)</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>(K, 0, R)</td>
<td>(K, 1, R)</td>
<td>(K, 2, R)</td>
<td>(F, ⊥, L)</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>(M, 0, L)</td>
<td></td>
<td></td>
<td>(Accept, ⊥, R)</td>
<td></td>
</tr>
</tbody>
</table>

All entries in the table not filled in are $(Reject, ⊥, R)$.

Explanation: In state Start, if we see a 2 or ⊥ right away, we reject (the input is not in the right format). Otherwise it remembers in the state the symbol that was in the first position and replaces it with ⊥.

**Shifting Right:** Using states $B, C, D, E$ it remembers the symbol being read, writes the symbol previously remembered and moves right until it hits ⊥. If the last symbol it saw was #, then $n = 0$ so it goes to a checking state (M) and moves left. If the last symbol it saw was 1 or 2, it can decrement the ternary number
by changing the last position to 0 or 1, respectively. Then it has decremented it, so it goes to state $H$ to scan left. Otherwise it saw a 0 and has not decremented, so it proceeds to state $F$, next.

**Decrement the Ternary Number:** In state $F$ it scans left to find the latest symbol in it that is not 0. If no symbol is not 0, then $n = 0$ so it goes to a checking state (M) and moves left. Otherwise it accomplishes a decrement by subtracting 1 from the current position, then replacing the rest of the string with 2s in state $G$, until it hits ⊥. Once that happens, it goes to state $H$.

**Scan Left for #:** In state $H$ it scans left until the #, then switches to state $I$ one position before the #.

**Decrement the Binary Number:** In state $I$ it scans left to find the latest symbol in it that is not 0. If no symbol is not 0, then $m = 0$ but we have already subtracted one from $n$ so they cannot be equal – reject. Otherwise it accomplishes a decrement by subtracting 1 from the current position, then replacing the rest of the string with 1s (in state $J$), until it hits the #. Once that happens, it goes to state $K$.

**Set up for Next Round of Decrements:** In state $K$ it scans right until ⊥, then moves left and changes to state $F$. As we have seen, this will cause it to start decrementing the ternary number.

**Check:** If we get to state $M$, we have found out that $n = 0$ so we need to check that $m = 0$. So state $M$ scans left through 0s until ⊥. If it sees anything else, reject. Otherwise $m = 0$ so $m = n$, and we accept (the only place we do).

That’s the end of the formal description and explanation.

Here is an alternate approach; we give a low level description of the Turing Machine $M$ that decides $L$. It can be turned into a formal description. Our strategy here is to turn both the binary and ternary representations into unary representation and then compare them directly. We use a three tape deterministic Turing machine for this purpose. It can easily be turned into a single tape Turing machine.

$M =$ “On input string $s$ on the first tape of the machine:

1. Scan $s$ from left to right to check if $s$ is of the form $w\#x$ where $w \in \{0,1\}^*$, and $x \in \{0,1,2\}^*$.

2. Move back to the leftmost position of the first tape.

3. If the first tape contains 0 keep moving to the right.

4. If the first tape contains # go to step 6. If the first tape contains 1, write a 1 on the second tape.

5. If the first tape contains # go to step 6. If the first tape contains 0 duplicate the content of the second tape (i.e. if there were $n$ 1’s on the second tape, write $n$ new 1’s on the right). If the first tape contains 1, duplicate the content of the second tape and add a 1 on the right. Repeat this step.

6. If the first tape contains 0 keep moving to the right.

7. If the first tape contains ⊥ go to step 9. If the first tape contains 1, write a 1 on the third tape. If the first tape contains 2, write two 1’s on the third tape.

8. If the first tape contains ⊥ go to step 9. If the first tape contains 0 triplicate the content of the third tape. If the first tape contains 1, triplicate the content of the third tape and add a 1 on the right. If the first tape contains 2, triplicate the content of the third tape and add two 1’s on the right. Repeat this step.

9. Move the write heads on second and third tape back to the leftmost position.

10. Match the number of ones in second and third tape. If they equal number of 1’s accept. Otherwise reject.”
Here is the DPDA that recognizes $L$.

Figure 1: DPDA for recognizing $L$
(3) Yes, \( \text{prefix}(L) \) is regular. The construction is similar to the problem 5 of homework 1. Let \( M \) be the DFA for the regular language \( L \). We create an NFA \( N \) for \( \text{prefix}(L) \) by first adding a new final state, then adding \( \epsilon \)-transitions from all the states, from where there is a directed path to any of the final states of \( M \), to the new final state.

(b) Yes, \( \text{prefix}(L) \) is context-free. Let \( G \) be the CFG for \( L \) in CNF. We construct a CFG \( G' \) for \( \text{prefix}(L) \) from \( G \) in the following way: add all the variables and rules of \( G \) to \( G' \). For each variable \( A \) in \( G \) add a variable \( A_p \) to \( G' \). For each rule \( A \rightarrow BC \) in \( G \), add the rules \( A_p \rightarrow A|B_p|BC_p \) to \( G' \) (This rule captures the recursive definition of prefix strings. Consider a parse tree with \( A \) at the root. All the prefixes of the strings that can be derived from \( A \) are either the strings derived from \( A \) themselves, or the prefixes of the strings that can be derived from right sub-tree of \( A \), rooted at \( B \), or the any string derived from \( B \) and concatenated with any prefix of any string that can be derived from \( C \)). Make \( S_p \) the start variable of \( G' \).

(c) Yes, \( \text{prefix}(L) \) is Turing-recognizable. Let \( T \) be the Turing machine that decides \( L \). We use \( T \) to build a Turing machine \( V \) that recognizes \( \text{prefix}(L) \).

\[
V = \text{"On input string } x:\n\]

1. Successively for strings of length 0, 1, 2, \dots choose all strings \( y \) in lexicographical order.

2. Run \( T \) on \( xy \). If \( T \) accepts \( xy \) then accept. Otherwise, continue with the next string."

(4) Given \( a^n \), we use the familiar division method to check if \( n \) is a prime, i.e. if any of the numbers 2, \ldots, \( n-1 \) divides \( n \), then \( n \) is not a prime, otherwise it is a prime. Here is the low level description of the two tape Turing Machine \( M \) that decides \( L \):

\[
M = \text{"On input string } w \text{ on the first input tape, and all blanks on the second tape:}
\]

1. Scan \( w \) from left to right to determine if it is a member of \( a^3^+ \) (i.e. check if it consists of 3 or more \( a \)'s) and reject if it is not. Otherwise return the head to the leftmost end of the tape.

2. Write a \( b \) on the first cell of the second tape, and do not move the head.

3. As long as the first tape contains an \( a \) and the second tape contains a \( b \), keep moving both the heads to the right. If the first tape contains an \( a \) but the second tape contains \( a \bigcup \) (blank symbols), then write a \( b \) in the second tape and move the head on the first tape to the right. If the first tape now contains a \( \bigcup \) then accept. Otherwise return both the heads to the leftmost position.

4. Match an \( a \) on the first tape with a \( b \) on the second tape (by putting a X mark), and move both the heads to the right. Keep on matching as long as the first tape contains an \( a \) and the second tape contains a \( b \). If both the tapes contain \( \bigcup \) then reject. If the first tape contains an \( a \) and the second tape contains a \( \bigcup \) then move the second tape to the leftmost position (and replace the X mark by \( b \)'s), and start matching again. If the first tape contains a \( \bigcup \) and the second tape contains a \( b \) then return both the heads to the leftmost position (and replace the X marks by \( a \)'s and \( b \)'s on the first and the second tape respectively). Go to step 3."

(5) Given the quaternary representation of a number we can easily obtain the binary representation of the same number by replacing each digit of the quaternary representation by its binary equivalent, namely replacing 0 by 00, 1 by 01, 2 by 10 and 3 by 11. For example, \( 123_4 \) is the quaternary representation of the number \( 27_{10} \). Replacing each digit of the the quaternary representation by its equivalent binary representation, we
obtain $011011_2$ which is the binary representation of the same number. In fact this works for any base that is a power of two, e.g. hex representation of a 32 bit integer by 8 hexadecimal digits: $0x1a7e34bc$.

So our strategy is to generate strings of the form $x#x^{rev}$ where $x$ is the quaternary representation of a number, and then convert the $x$ before the # to the equivalent binary representation. The following grammar does exactly that:

\[
S \rightarrow A_0S0|A_1S1|A_2S2|A_3S3|# \\
A_0 \rightarrow 00 \\
A_1 \rightarrow 01 \\
A_2 \rightarrow 10 \\
A_3 \rightarrow 11
\]