1 Problem 1

1.1 Simplifications to the machine

We can assume that the machine only accepts with empty stack tape. If it isn't the case, and we had some accepting state S, we can add the following transitions. When the machine is in state S, and the stack isn't empty, then it pops from the stack. If the machine is in state S, and the stack is empty (is reads $\perp$), then it moves to state S', which is the new accepting state.

1.2 Idea of the solution

Let's define *extended state* of the TM in the problem as a tuple (head position on the input tape, state, whole content of the working tape with its head position) (note that we omit stack content). Obviously, the number of possible extended states is polynomial, since memory apart from the stack is logarithmic.

Assuming that we finish with an empty stack, when we push something onto the stack, there must be a moment when we pop it (meaning the first moment when we are on this position, and we move left).

The sequence of state changes in the TM between putting an element on the stack, and deleting it from the stack for the first time, doesn't depend on what's already on the stack. This is because during this period, the program can't move its head on the stack tape to see the characters that were pushed onto the stack earlier and take them into consideration. It means this sequence of states only depends on the rest of the state of the TM.

In our solution, we *accelerate* the machine by *remembering* for every pair (extended state, character pushed down onto the stack) the state the machine would end up in after removing the considered character from the stack.

We keep in memory a mapping: from (extended state, character) pair to the new state after popping this character from the stack. If we already pushed some (extended state, character), but we haven't popped it yet we also remember such states this in the mapping. Additionaly instead of just keeping on character A in the stack we will keep a pair (A, encoded state), so we can fill the mapping described above.

The algorithm on our new machine works as follows:

- If we do not move the stack head in the original machine, we do exactly the same move as the original machine.
- If we move the stack head to the right writing A, and the pair (current extended state, A) has a saved *popping* move, then instead of pushing A onto the stack we just move to the respective state.
- If we move the stack head to the right writing A, and there is no saved *popping* move for (current extended state, A), but we already pushed this exact pair onto the stack (but we haven't popped it yet, we can also check this in the mapping), then it means that the original TM loops. In this case we reject the word (or loop if we want to exactly emulate the original machine).
- If we move the stack head to the right writing A, there is no saved *popping* move for (current extended state, A), and we haven't pushed this pair to the stack yet, we do the same move as the original machine but pushing (extended state, A) onto the stack instead of A.
- If we move the stack head to the left, we also pop the encoding of the original state we pushed A with, and fill the mapping from this state and A to the new state we're now going to. Then we proceed with the move.

In the algorithm above we assume that we can encode the extended state in some way, which can obviously be done.

1.3 Mapping

We keep in memory a mapping from (state, input head position, content of the log-space working tape, working tape head position, top of the stack) to (new state, new input head position, new content of the log-space working tape, new working tape head position) after we pop this element from the stack.

We consider this tape as RAM – even more, we assume that we can find some element with an unknown position efficiently. This only adds a polynomial factor to the complexity of our TM (we assume that every time we read some mapping, we traverse the whole tape), because the size of the mapping is polynomial (the number of possible states is polynomial, and the size of an encoded state is logarithmic, as we need to keep 3 numbers, log characters, and one additional character).

1.4 Complexity of the new machine

Since the pairs (character on the top of the stack, extended state) encountered by the new machine when pushing to its stack are distinct (otherwise we detect that the original TM loops), the stack would always have polynomial size. Moreover, we already explained why the mapping has polynomial size. Altogether, the space complexity is polynomial (it is also true for the original machine).

Note that the time complexity of the original machine can be even exponential.

In our algorithm, we never push the same character with the same state onto the stack twice (not only two of them being at the stack at the same time as in the space argument). This is because if we have ever pushed this character with this state onto the stack, then we either looped, or we already know the *popping* move, so we won't push it onto the stack again.

Summing up, our new machine works in polynomial time.

2 Problem 2

2.1 Prerequisitions

We know that for every language in P there exists a Turing machine, which for given n can produce a boolean circuit which recognizes if a word of length n is in the language. We will use this machine to build a machine with given restrictions.

2.2 Idea

We will have the machine producing the boolen circuit online and use it just as we would be able to fit the circuit into the memory.

Then we will simulate the binary operations in boolean circuits using the stack for the gates for which we started computing the gates, on which they depend, but not computed the gate itself.

In the end we will compute gates multiple times, but we don't care about the time.

2.3 Boolean circuits

We can assume that the boolean circuit, which is produced has only 2 input to the gates (if not, then we can change it to have 2 input gates increasing the depth by logarithmic factor building a binary tree).

We can also add numbering to the states, binary representation of these numbers won't be longer than logarithmic, because the circuits have polynomial size.

We can't fit the whole circuit into the working memory, but we can keep a Turing machine producing this circuit and order this machine to run and print some small subset of output (which we want to read, eg. one character in the extreme). So by multiplying runtime by (very large) polynomial factor we can assume that we can read the boolean circuit representation.

2.4 Note to reading more than one character from the stack

If we want to read (match) more than one character from the stack we can always copy it to the working memory and then read it freely (without the need to overwrite it with $\perp$). Note that number of the gate takes only logarithmic memory, so we meet this restriction.

2.5 Computing the boolean circuit

In order to compute whether the word belongs to the language we have to compute the value of the ouput gate.

We would keep on the stack triples: (g, v_1, v_2). They represent number of the gate we want to compute and the values that we already computed.

To compute a gate value we put $(g, \perp, \perp)$ on the stack.

Our algorithm has two steps which we repeat until the stack isn't empty. We start with the looking step.

Looking step:

- If we see $(c, \perp, \perp)$ at the top of the stack and c is an input gate, then we pop this from the stack and proceed to the propagating step keeping value of the c in the working memory (which we can read from the input)
- If we see $(g, \perp, \perp)$ then we read the number of the first input for this gate – g_1 and push $(g_1, \perp, \perp)$ onto the stack and proceed to looking step
- If we see $(g, v_1, \perp)$ then we read the number of the second input for this gate – g_2 and push $(g_2, \perp, \perp)$ onto the stack and proceed to looking step
- If we see (g, v_1, v_2) then we evaluate g ($g = v_1 \wedge v_2$ or $g = v_1 \vee v_2$) we pop this from the stack and proceed to the propagating step keeping value of g in the working memory

Propagating step: We assume that we have val kept in our working memory and after this step we always go to the looking step

- If the stack is empty we have the value of our original gate computed – it's val
- If we see $(g, \perp, \perp)$ at the top of the stack then we change it into $(g, val, \perp)$
- If we see $(g, v_1, \perp)$ at the top of the stack then we change it into (g, v_1, val)
- It's impossible to see (g, v_1, v_2) pattern in this state (because after seeing such pattern we never push to the stack) and propagating step only appears after popping something from the stack)

2.6 Proof

It's obvious that is uses logarithmic memory. As for correctness the inductive proof is quite obvious. At every moment we keep some correct partial computation and it will halt some time.