4 Undecided?

(a) nk

Since we're told, for this part specifically, to consider the machine has n (different) states and the algorithm has k (different) instructions, so there are nk-many different state-instruction combinations.

We first observe that each of the j returned by the algorithm $\mathscr{A}$ correponds to a distinct instruction. So, I'll denote a single output of $\mathscr{A}$, which is a machine state c and a number j, as a state-instruction combination.

Since each different state-instruction combination would lead to a new computation, and also since two identical state-instruction combination would lead to repeated computation, so an algorithm of k instructions can perform a maximum of nk-many iterations on an n-state machine without repeating ay computation.

This is possible by having each of these iterations returning a new state-instruction combination (a previously unused pair of instruction number j and machine state c) that hasn't been computed before. In other words, consider the rotational situation where each of the computations $i_j(s_k)$ would return (i_j, s_{k+1}) for all $1 \leq k < n$, and $i_j(s_n)$ would return (i_{j+1}, s_1) for all $0 \leq j < k-1$, and finally, let $i_{k-1}(s_n)$ return (i_0, s_1), the first combination. Here, no repeating computation exists for the first nk iterations of algorithm $\mathscr{A}$.

(b) Direct Proof

We proceed by first showing that if the algorithm is still running after $nk+1$ iterations, it will loop forever.

As proved in part (a), the maximum number of iterations that doesn't repeat any computation is nk, which implies that if the algorithm $\mathscr{A}$ is still running after $nk+1$ iterations, then by the Pidgeonhole Principal, $\mathscr{A}$ must have repeated computations somewhere.

Let the repeated computation be at these two iterations k_i, k_j where $1 \leq k_i < k_j \leq nk+1$. So the computations at ${k_i}^{th}$ and ${k_j}^{th}$ are repeated, or the same. Then, consider the set of $(j-i)$ iterations $S = \{k_i, k_{i+1}, ..., k_{j-1}\}$. This would be a cycle as we reach the ${k_j}^{th}$ iteration, which means that the set of $(j-i)$ iterations $S' = \{k_j, k_{j+1}, ..., k_{2j-i-1}\}$ is exactly the same as the set S, thus indicating that a loop must have occured, which implies that the algorithm $\mathscr{A}$ would loop forever if the algorithm is still running after $nk+1$ iterations.

Then, since $n, k \in \mathbb{Z}^+$, so $2n^2k^2 = n^2k^2 + n^2k^2 \geq nk+1$, and thus, the algorithm $\mathscr{A}$ would loop forever if the algorithm is still running after $2n^2k^2$ iterations, as desired.

Q.E.D.

(c) No, it doesn't.

Algorithm designed using our results from part (a) and (b):

$Halts\_Here(\mathscr{A}, x)$:
  if algorithm $\mathscr{A}$ halts on input x within $(nk+1)$ iterations, then return "yes"
  if algorithm $\mathscr{A}$ does not halt on input x within $(nk+1)$ iterations, then return "no"

This does not contradict the undecidability of the Halting problem because it's checking whether algorithm $\mathscr{A}$ halts within a finite number (nk) of iterations, where the Halting problem isn't, which makes the difference. Moreover, we actually proved in Discussion that checking whether a program halts within a finite number of iterations is actually computable.