Q2:

In order to show that P(x) is computable in S\_n ,we give a turing machine M, the decides P(x).

For any input of the form s_{i_1}, s_{i_2}, ... , s_{i_k} , the turing machine M works as follows:

* It starts in state s0.
* At everys string reads it switches its state from s0 to s1 and s1 to s0.
* If after reading the complete input it is in state s0, then it accepts the string.
* Else rejects.

Since after reading the input if the TM is in s0 state, then the length is even and hence the string is accepted.

Q3:

The halting problem is a decision problem about properties of computer programs on a fixed Turing-complete model of computation, i.e., all programs that can be written in some given programming language that is general enough to be equivalent to a Turing machine. The problem is to determine, given a program and an input to the program, whether the program will eventually halt when run with that input. In this abstract framework, there are no resource limitations on the amount of memory or time required for the program's execution; it can take arbitrarily long and use an arbitrary amount of storage space before halting. The question is simply whether the given program will ever halt on a particular input.

For example, in pseudocode, the program

while (true) continue

does not halt; rather, it goes on forever in an infinite loop. On the other hand, the program

print "Hello, world!"

does halt.

While deciding whether these programs halt is simple, more complex programs prove problematic. One approach to the problem might be to run the program for some number of steps and check if it halts. But if the program does not halt, it is unknown whether the program will eventually halt or run forever. Turing proved no algorithm exists that always correctly decides whether, for a given arbitrary program and input, the program halts when run with that input. The essence of Turing's proof is that any such algorithm can be made to contradict itself and therefore cannot be correct.

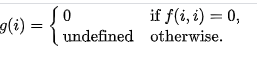
The concept above shows the general method of the proof; this section will present additional details. The overall goal is to show that there is no total computable function that decides whether an arbitrary program *i* halts on arbitrary input *x*; that is, the following function *h* is not computable (Penrose 1990, p. 57–63):

{\displaystyle h(i,x)={\begin{cases}1&{\text{if }}{\text{ program }}i{\text{ halts on input }}x,\\0&{\text{otherwise.}}\end{cases}}}

Here *program i* refers to the *i* th program in an enumeration of all the programs of a fixed Turing-complete model of computation.

The proof proceeds by directly establishing that no total computable function with two arguments can be the required function *h*. As in the sketch of the concept, given any total computable binary function *f*, the following partial function *g* is also computable by some program *e*:Possible values for a total computable function *f* arranged in a 2D array. The orange cells are the diagonal. The values of *f*(*i*,*i*) and *g*(*i*) are shown at the bottom; *U* indicates that the function *g* is undefined for a particular input value.

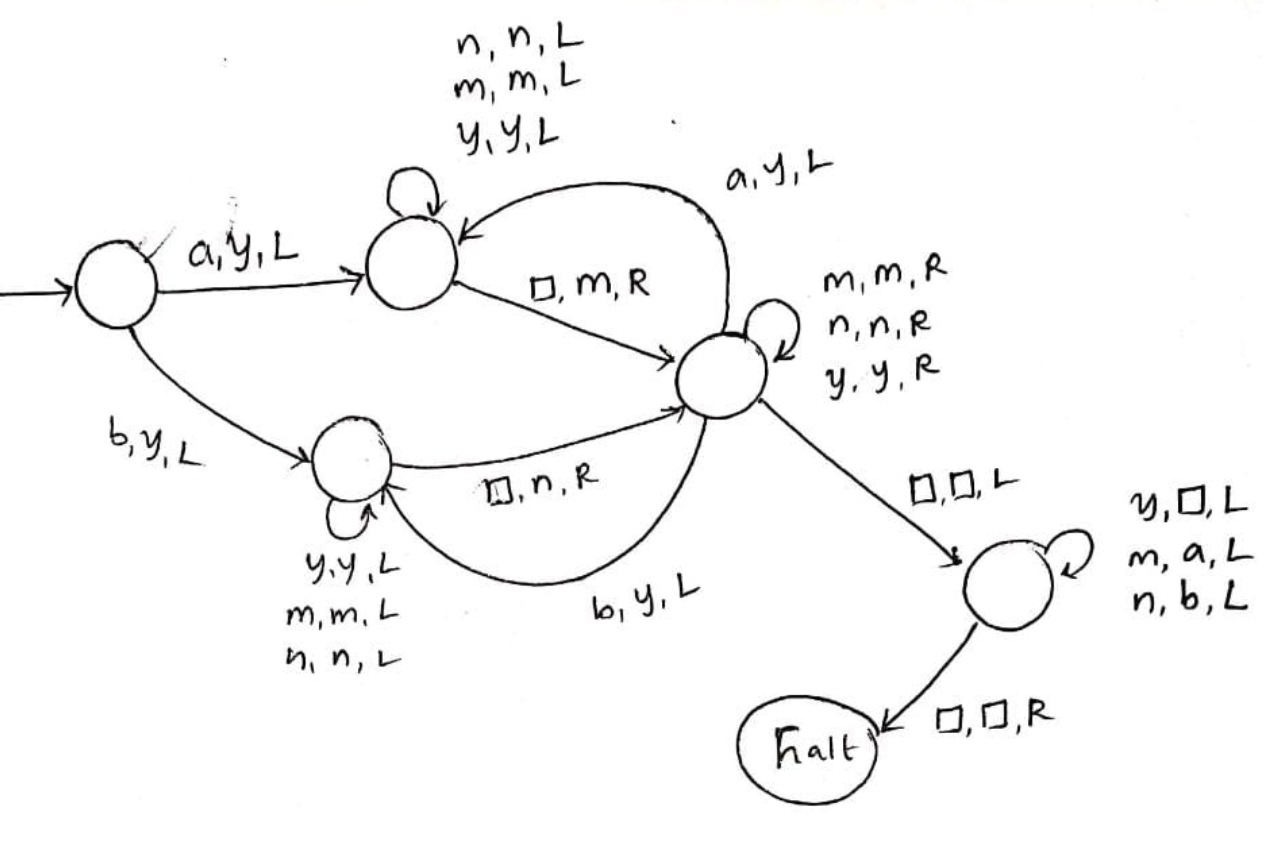
{\displaystyle g(i)={\begin{cases}0&{\text{if }}f(i,i)=0,\\{\text{undefined}}&{\text{otherwise.}}\end{cases}}}



The verification that *g* is computable relies on the following constructs (or their equivalents):

* computable subprograms (the program that computes *f* is a subprogram in program *e*),
* duplication of values (program *e* computes the inputs *i*,*i* for *f* from the input *i* for *g*),
* conditional branching (program *e* selects between two results depending on the value it computes for *f*(*i*,*i*)),
* not producing a defined result (for example, by looping forever),
* returning a value of 0.

Q5:



***Explanation***

top part is to write m for one a

lower part is to write n for one b

Changing all m's and n's back to a and b respectively