Theory of Computation

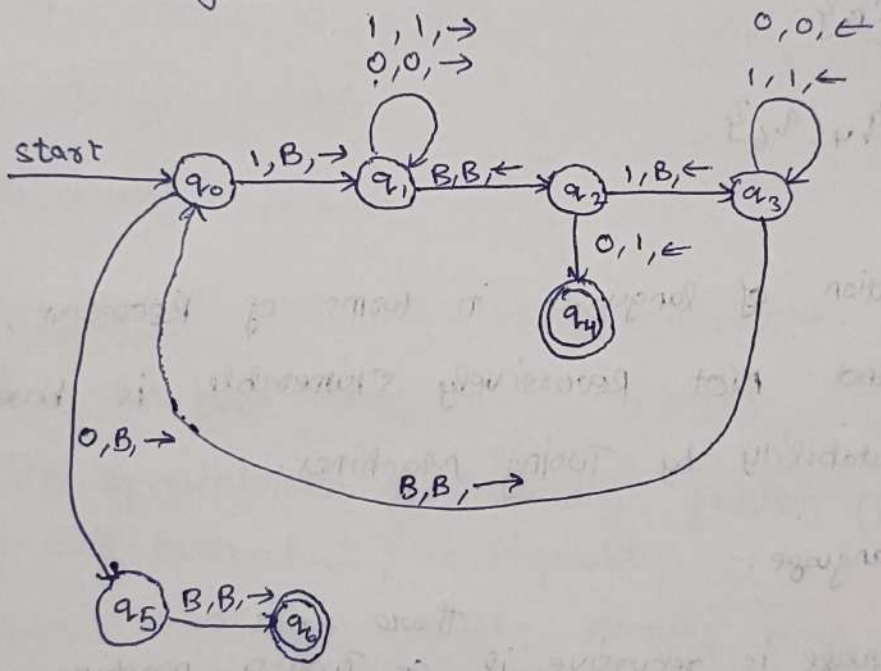
Assignment 12

Name : Vishal . K

Reg. No : 312222500161

1. Turing Machine for performing unary subtraction:

Transition Diagram:



Transition function:

$\delta(q_0, 1) = (q_1, B, \rightarrow)$

$\delta(q_1, 0) = (q_1, 0, \rightarrow)$ $\quad \delta(q_1, 1) = (q_1, 1, \rightarrow)$

$\delta(q_1, B) = (q_2, B, \leftarrow)$

$\delta(q_2, 0) = (q_4, 1, \leftarrow)$ $\quad \delta(q_2, 1) = (q_3, B, \leftarrow)$

$\delta(q_3, 0) = (q_3, 0, \leftarrow)$ $\quad \delta(q_3, 1) = (q_3, 1, \leftarrow)$

$\delta(q_3, B) = (q_0, B, \rightarrow)$

$\delta(q_0, 0) = (q_5, B, \rightarrow)$

$\delta(q_5, B) = (q_6, B, \rightarrow)$

Turing Machine:

$TM = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

$Q \rightarrow \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$

$\Sigma \rightarrow \{0, 1\}$

$\Gamma \rightarrow \{0, 1, B\}$

$q_0 \rightarrow q_0$

$B \rightarrow \{B\}$

$F \rightarrow \{q_4, q_6\}$

3. The classification of languages in terms of Recursive, Recursively Enumerable and Not Recursively Enumerable is based on their computability by Turing Machines.

Recursive Language:

* A language is recursive if there exists a Turing machine always halts and correctly decides whether an input belongs to the language.
* Eg: Checking if a program has valid syntax according to the Programming language.
* Real world Application: often used in compilers and static code analysing tools like vscode Intellisense.

Recursively Enumerable Language:

* A language is Recursively enumerable if a Turing machine accepts all valid input (with respect to that language) but may (or may not) loop forever for other inputs.
* Eg: The Halting Problem is one such problem (whether a program ends for an given input).
* Real world application: used in program verification, where detecting errors is possible, but proving its correctness is not always possible.

Not Recursively Enumerable:

* A language is not recursively enumerable if no Turing Machine can even recognize its elements.
* Eg: The complement of the halting problem (proving a program will never halt) is impossible.
* Real World Application: Absolute security proofs (in cryptography) are mostly not RE, as there is no algorithm to verify all possible vulnerabilities.

Undecidable but RE Languages:

* A language is classified as Undecidable but RE if there exists a Turing machine that can recognize valid inputs but cannot decide for all input cases.

Eg: In the halting problem, if a problem halts, we can detect it, but we cannot decide the same for all cases.

* Real World Application: AI generated content verification (detecting bias is possible, but proving all outputs are unbiased is not decidable).

4. w_1 w_2 w_3 w_4 w_5
List A: 10, 101, 011, 0, 110

List B: 101, 01, 1, 01, 110
x_1 x_2 x_3 x_4 x_5

Since there are two operations namely 01 and 01 in list B, it probably suggests that repetition of the same operation is not ~~allowed~~ allowed within a list.

If that is the case, List A (having 12 characters) and List B (having 11) can never be ordered such that they give the same result.

Let us assume that repetition is allowed. Let us try mapping

	w_1 w_2	w_3	w_4 w_2	w_5
A	10,101	011	0,101	110
↕	↓	↓	↓	↓
B	101,01	01,1	01,01	110
	x_1 x_2	x_2 x_3	x_2 x_4	x_5

All ops in A can be mapped to ops in B. Also

B A

1,01 ⟶ 101.

Hence the 1 in B can also be mapped to A.
Hence both A & B can be mapped to each other

$w_1 w_2 w_3 w_4 w_2 w_5$

$\equiv$

$x_1 x_2 x_2 x_3 x_2 x_4 x_5$

Such a transformation lead to the same state.
however even the above mapping would be invalid if order of operations is to be maintained.

Even if repetition of operations is not allowed, both A & B lead to the same final state if it is not mandatory to include all operations from each list. One such example mapping is given below.

	w_1	w_2	w_5
A :	10	101	110
	↕		↕
			110
B :	101	01	
	x_1	x_2	x_5